About This Book

This book introduces you to the programming tools and services for writing an application program using an IBM RT™ system. It provides an overview of the programming process, and describes how to use RT programming tools and interfaces within that programming process. It includes information about:

- What the system structure is
- Designing output for the display
- C language programming tools
- Program maintenance and source code control
- Using system calls
- Using Library Functions
- Installing a program on the system
- Using the trace and error logging facilities
- Writing messages.

What You Should Know

The book uses the C programming language in the examples, and many of the tools work only with C language source files. Therefore, you should be familiar with the C programming language to get the most out of this book. However, programmers working with other high level languages can also benefit from the information in this book. In addition to knowing the C language, you should:

- Have experience in writing application programs.
- Be able to use the RT system to:
 - Enter commands
 - Create and delete files
 - Edit files
 - Move around in the file system.

RT is a registered trademark of International Business Machines Corporation.

Related Information

To become familiar with the operating procedures of the RT system, refer to the following book(s):

- *IBM RT Using the AIX Operating System*
- *IBM RT Managing the AIX Operating System*
- *IBM RT Using DOS Services*

To help understand the information in this book, and to provide more detail about the commands, calls and library functions introduced in this book, refer to the following books:

- *IBM RT AIX Operating System Commands Reference*
- *IBM RT AIX Operating System Technical Reference*
- *IBM RT C Language Guide and Reference*
- *Japanese Language Support User's Guide*

To write device drivers for use on the RT system, refer to the following books:

- *IBM RT Device Driver Development Guide*
- *IBM RT AIX Operating System Technical Reference*

The programs described in this book generate messages when errors occur. Refer to the following book for explanation of messages that the system programs generate:

- *IBM RT Messages Reference*

A Reader's Comment Form and Book Evaluation Form are provided at the back of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

Ordering Additional Copies of This Book

To order additional copies of this publication (without licensed program diskettes), use either of the following sources:

- To order from your IBM representative, use Order Number SBOF-1819.
- To order from your IBM dealer, use Part Number 27F4363.

Ordering either of these numbers includes the following items:

- Book
- Binder
- Slipcase
- Example program diskette.

For information on ordering the binder or book (with example program diskette) separately, contact your IBM representative or your IBM dealer.

IBM RT Advanced Interactive Executive Operating System Version 2.2.1

AIX Operating System Programming Tools and Interfaces



SC23-2016-1

Second Edition (September 1988)

Portions of the code and documentation described in this book were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California under the auspices of the Regents of the University of California.

This edition applies to Version 2.2 of the IBM AIX Operating System. Changes are made periodically to the information herein; these changes will be reported in technical newsletters or in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your IBM authorized RT dealer, your IBM marketing representative, or your IBM authorized remarketer.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

IBM is a registered trademark of International Business Machines Corporation
AIX is a registered trademark of International Business Machines Corporation
© Copyright International Business Machines Corporation 1985, 1987, 1988
© Copyright INTERACTIVE Systems Corporation 1984
© Copyright AT&T Technologies 1984

Contents

Chapter 1. Programming with RT 1-1
About This Chapter 1-2
Programming Tools 1-3
Shared Libraries 1-5
Programming Interfaces 1-9
Using the Programming Examples 1-11

Chapter 2. Compiling and Linking Programs 2-1
About This Chapter 2-2
Compiling a Program 2-3
Checking C Programs 2-5
Other C Programming Tools 2-19
Processing Assembler Language Routines 2-20
Building Programs with make 2-22

Chapter 3. Using the Subroutine Libraries 3-1
About This Chapter 3-2
System Libraries 3-3
The C Library 3-8
Run-Time Services Library 3-30
Math Library 3-32
Network Management Library 3-34

Chapter 4. Using System Calls 4-1
About This Chapter 4-2
Header Files Needed for Calls 4-2
Process Calls 4-4
Interprocess Communications 4-29
System Memory Management 4-67
File System Calls 4-78
Time System Calls 4-84
Using System Calls with Distributed Services 4-85

Chapter 5. Controlling the Terminal Screen 5-1
About This Chapter 5-2
Introduction 5-3
Using the Library Routines 5-12
Routines for Panels and Panes 5-21
Display Attributes 5-27

Using Other Features 5-33
Example Program 5-36

Chapter 6. Writing Messages and Help 6-1
About This Chapter 6-2
Messages 6-3
Building a Message Table 6-8
Using Messages in a Program 6-13
Using Variable Fields in Message Text 6-15
Help 6-22
Building a Help File 6-25
Using Help in a Program 6-29

Chapter 7. Monitoring Program Activities 7-1
About This Chapter 7-2
Overview 7-3
Using the Trace Facilities 7-4
Using the Error Log Facilities 7-23
Using the Dump Facilities 7-42

Chapter 8. Debugging Programs 8-1
About This Chapter 8-4
The dbx Symbolic Debugger 8-5
dbx Command Summary 8-6
Using dbx with X-Windows 8-37
The sdb Program 8-47
sdb Command Summary 8-48
Using the Program 8-51
Displaying and Manipulating the Source File 8-55
Controlling Program Execution 8-57
Debugging Assembler Language 8-60
An Example of a Debug Session 8-61
The adb Program 8-64
Displaying Instructions and Data 8-69
Debugging Program Execution with adb 8-88
Using the adb Memory Maps 8-100
Combining Commands on a Single Line 8-102
Creating adb Scripts 8-103
Setting Output Width 8-107
Setting the Maximum Offset 8-107
Setting Default Input Format 8-108
Using AIX Operating System Commands 8-108
Computing Numbers and Displaying Text 8-109
An Example: Directory and Inode Dumps 8-110
Patching Binary Files 8-113

Chapter 9. Installing and Updating a Program 9-1
About This Chapter 9-2
Understanding System Guidelines 9-3
Using Installation and Update Services 9-6
What You Need to Install a Program 9-9
What You Need to Update a Program 9-15
Installing and Updating Active Files 9-21
Allowing for Recovery 9-24
Creating the Program History File 9-26
Creating the Program Requirements File 9-29
Creating the Program Name File 9-33
Creating an Apply List File 9-34
Creating an Archive Control File 9-35
Creating a Special Requirement File 9-36
Creating a Save and Recover Directory 9-39
Designing Programs for a Code-Service Environment 9-43

Chapter 10. Maintaining Different Versions of a Program 10-1
About This Chapter 10-2
Introducing SCCS 10-3
Using SCCS Commands 10-8

Chapter 11. Finding and Changing Strings 11-1
About This Chapter 11-2
Finding Strings 11-3
Scanning Files 11-6
Editing Files with sed 11-18

Chapter 12. Using the Macro Processor (m4) 12-1
About This Chapter 12-2
Using the Macro Preprocessor 12-3
Defining Macros 12-4
Using Other m4 Macros 12-8

Chapter 13. Creating an Input Language 13-1
About This Chapter 13-3
Writing a Lexical Analyzer Program with lex 13-4
The lex Specification File 13-6
Regular Expressions 13-8
Actions 13-14
Passing Code to the Generated Program 13-19
Defining Substitution Strings 13-20
Start Conditions 13-21
Compiling the Lexical Analyzer 13-22

Using lex with yacc 13-23
Creating a Parser with yacc 13-25
Grammar File 13-26
Using the Grammar File 13-28
Declarations 13-31
Rules 13-34
Actions 13-36
Programs 13-39
Error Handling 13-40
Lexical Analysis 13-43
Parser Operation 13-44
Using Ambiguous Rules 13-47
Turning on Debug Mode 13-50
Creating a Simple Calculator Program - Example 13-51

Chapter 14. International Character Support 14-1
About This Chapter 14-2
Introduction 14-3
Kernel Modifications 14-8
Environment Variables 14-10
Character Display and Handling 14-12
String Handling 14-15
Intersystem Compatibility 14-16
Limits to Support 14-20
Collation 14-21
Files and Directories 14-22

Chapter 15. Using Remote Procedure Call (RPC) 15-1
About This Chapter 15-2
Overview of RPC 15-3
RPC Programming Levels 15-18
Authenticating Remote Procedure Calls 15-30
Special Topics 15-35
Example Applications 15-45

Chapter 16. Message Facility 16-1
Message Catalogs 16-1
Message Text Source Files 16-3
Creating Catalogs 16-10
Using Catalogs 16-10

Appendix A. Installing Programming Examples A-1

Appendix B. Extended curses Structures B-1

Appendix C. RT Printer Support Data Stream C-1
Using Printers from A Program C-2

Appendix D. ASCII Characters D-1

Appendix E. Porting 4.3BSD Programs E-1

Glossary X-1

Index X-21

Figures

2-1. Example lint Library Input File 2-17
2-2. Rules for Creating Files 2-30
2-3. Example Default Rules File 2-31
2-4. Example Description File 2-45
3-1. Summary of System Libraries 3-4
3-2. Comparison of I/O Operations 3-10
3-3. Comparison of Japanese Language Support I/O Operations 3-11
4-1. Using the Fork System Call 4-7
4-2. Example of Fork System Call 4-8
4-3. Output from forktst1 Program 4-9
4-4. Fork and Wait System Calls - Sample Program 4-11
4-5. Output from forktst2 Sample Program 4-12
4-6. Exec System Call - Sample Program 4-13
4-7. Output from forktst3 Sample Program 4-14
4-8. Using the pipe System Call 4-15
4-9. Relationship of IDs in the System 4-20
4-10. Example of Process ID System Calls 4-24
4-11. Output for Process ID System Call Program 4-25
4-12. User Controlled Signals 4-30
4-13. Example of Signal Trapping 4-33
4-14. Enhanced Signal Calls 4-34
4-15. Sources of Signals 4-35
4-16. Signal Responses 4-36
4-17. sigvec Structure Members 4-37
4-18. Enhanced Signals Example Program 4-40
4-19. Semid Data Structure 4-43
4-20. Semaphore Structure 4-43
4-21. Semop System Call Parameters 4-44
4-22. How sem_op Specifies a Semaphore Operation 4-45
4-23. Using Semaphores Concept Example 4-46
4-24. Semop Call for Proca 4-47
4-25. Semop Call for procb 4-48
4-26. Semaphore Usage 4-49
4-27. Using Semaphore Calls 4-50
4-28. Message Queue Terms 4-56
4-29. Example of Using Message Queues 4-63
4-30. Segment Register Addressing 4-68
4-31. Segment Register Usage 4-69
4-32. Shared Memory Terms 4-75
4-33. Example Program Listing for distserv.c 4-88

4-34. Output from Example Program distserv.c 4-92
4-35. Example Program Listing for locktest.c 4-95
4-36. Output from Example Program locktest.c 4-99
4-37. Message Queue Creation Example Program 4-101
4-38. Send Message Example Program 4-103
4-39. Receive Message Example Program 4-105
4-40. Message Queue Creation Example Program Output 4-108
4-41. Message Queue Already Exists Example Program Output 4-108
4-42. Send Message Example Program Output 4-108
4-43. Receive Message Example Program Output 4-109
5-1. Screen Coordinate Boundaries 5-8
5-2. Example Panel Final Appearance 5-24
5-3. Creating Panes in the Panel 5-24
5-4. Links in the Panel and Pane Structure 5-25
5-5. Program to Create Example Panel 5-26
5-6. Display Attributes 5-28
5-7. Changing Screen Attributes Example Program 5-31
5-8. Control Codes 5-33
5-9. Example of Extended curses Program 5-37
6-1. Message Fields 6-4
6-2. System Identifiers 6-6
6-3. Content of Message Standard Format File 6-9
6-4. Header Files 6-13
6-5. Standard Symbols 6-15
6-6. Example of Integer Symbol Programming 6-17
6-7. Example of Character String Symbol Programming 6-19
6-8. Example of Text Insert Symbol Programming 6-21
6-9. Content of Help Text Format File 6-26
7-1. Trace Components 7-5
7-2. Example Program Fragment Showing Use of trcunix Subroutine. 7-13
7-3. Example Program Fragment Showing Use of trsave Subroutine. 7-15
7-4. Trace Template Syntax 7-17
7-5. Fields in a Trace Template 7-18
7-6. Example of a Trace Template for hook ID 330 7-21
7-7. Example of Output from the Trace Formatter 7-22
7-8. Error Components 7-24
7-9. Example Program Fragment Using errunix Subroutine. 7-32
7-10. Example of a Program Fragment Showing Use of errsave Subroutine. 7-34
7-11. Error Template Syntax 7-36
7-12. Fields in an Error Template 7-37
7-13. Example of an Error Template for Error Entries with Error ID 613 7-40
7-14. Example Output from the Error Formatter 7-41
7-15. Dump Components 7-43
7-16. Example of Output from the Dump Formatter 7-46
8-1. Example of Debug Session 8-62
8-2. Program Listing for adbsamp.c 8-70

8-3. Registers Displayed by adb 8-82
8-4. Program Listing for adbsamp2.c 8-88
8-5. Tracing Multiple Functions Example Program 8-97
8-6. Example adb Script File 8-103
8-7. Example adb Script File Output 8-104
8-8. Example Directory Dump Output 8-111
9-1. Install and Update Commands 9-6
9-2. Internal Commands 9-8
9-3. Return Codes to installp 9-13
9-4. Return Codes to updatep 9-18
9-5. Example of update Procedure Code 9-20
9-6. Record Format for the History File 9-26
9-7. Fields in a History Record 9-27
9-8. Record Format for Requirements File 9-29
9-9. Fields in Requirements Record 9-29
9-10. Entry Format for Program Name File 9-33
9-11. Fields in Program Name Entry 9-33
9-12. Requirements File Parameters 9-36
9-13. Save/Restore Directory Content 9-39
10-1. Parts of an SID 10-5
10-2. Growth of an SCCS File with Branching 10-5
10-3. Example of Using SCCS to Create and Update a File 10-8
10-4. Growth of an SCCS File with Branching 10-14
11-1. awk Special Characters 11-10
11-2. sed Command Flags 11-20
11-3. sed Block Diagram 11-20
11-4. sed Wildcard Characters 11-22
11-5. Syntax Symbols 11-23
11-6. sed Command Summary 11-24
12-1. m4 Built-in Macros 12-8
13-1. Simple Finite State Model 13-5
13-2. Regular Expression Operators 13-8
13-3. Special Characters 13-11
13-4. lex with yacc 13-24
13-5. yacc Literal Strings 13-29
13-6. yacc Grammar File for Calculator Program - calc.yacc 13-53
13-7. lex Specification File for Calculator Program - calc.lex 13-58
14-1. Code Points 14-8
15-1. How to Assign Program Numbers 15-5
15-2. Using the rnusers Library Routine in a Program 15-19
15-3. Using callrpc to Determine Number of Remote Users 15-21
15-4. Using registerrpc to Register RPC Calls with a Server 15-22
15-5. Using svc_register, a Low-Level RPC Routine 15-23
15-6. Low-Level RPC Client Passing an RPC Program using UDP/IP 15-27
15-7. Example of Authentication Credentials Structure 15-30
15-8. Remote Users Service Example 15-33

15-9. Code for the Library Routine svc_run 15-35
15-10. Using RPC Batching with TCP/IP Delivery 15-39
15-11. Example of a Client Using Batching to Create Strings 15-42
15-12. A Server Supporting Two Versions of the Same Program 15-45
15-13. C Procedure Handling Two Versions of the Same Program 15-46
15-14. Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP) 15-48
15-15. Example of Using a Callback Procedure 15-53
15-16. Using the gettransient Routine with a Client 15-55
15-17. Using the gettransient Routine with a Server 15-57
B-1. Structure Definition for WINDOW B-2
B-2. Structure Definition for PANEL B-5
B-3. Structure Definition for PANE B-7
C-1. Printer Control Codes C-3
D-1. Code Page 0 D-2
D-2. Code Page 1 D-3
D-3. Code Page 2 D-4

Chapter 1. Programming with RT

CONTENTS

About This Chapter 1-2
Programming Tools 1-3
 Entering a Program 1-3
 Checking a Program 1-3
 Compiling and Linking a Program 1-4
 Correcting Errors in a Program 1-4
 Building and Maintaining a Program 1-4
Shared Libraries 1-5
 Creating a Shared Library 1-5
 Shared Library Keys 1-6
 Installing Shared Libraries 1-6
 Linking Shared Libraries 1-8
 Changing Shared Libraries 1-8
Programming Interfaces 1-9
 Shell Commands 1-9
 Library Routines 1-10
 System Calls 1-10
Using the Programming Examples 1-11

About This Chapter

This chapter describes the IBM RT tools and services for developing application programs. In addition, it indicates where to get more information about these facilities, both in this book and in other RT system books.

Programming Tools

The RT system has many tools to help develop a C language program. To access any of these tools, enter a system command on the command line. These tools provide help in the following programming areas:

- Entering a program into the system
- Checking a program
- Compiling and linking a program
- Correcting errors in a program
- Filing and maintaining a program.

Entering a Program

The system has a line editor to help enter a program into a file to be compiled. The editor is called **ed**. Refer to *Using the AIX Operating System* for instructions about how to use this editor.

In addition, the system has two full screen editors, INed[1] and **vi**. These editors display a full screen of data and allow interactive editing of the file.

Checking a Program

The following programs help check the format of a program for consistency and accuracy:

lint Checks for syntax, data type and other programming and usage errors. Refer to "Checking C Programs" on page 2-5 for information about using this program.

cflow Generates a flow diagram of a C language program. Refer to "Other C Programming Tools" on page 2-19 for information about this program.

cxref Generates a cross reference listing for a C language program. Refer to "Other C Programming Tools" on page 2-19 for information about this program.

cb Reformats a C language source program into a consistent, indented format. Refer to "Other C Programming Tools" on page 2-19 for information about this program.

1 INed is a trademark of INTERACTIVE Systems Corporation

Compiling and Linking a Program

The **cc** command compiles and links a C language program with one command line entry. Refer to Chapter 2, "Compiling and Linking Programs" on page 2-1 for information about using this program.

Correcting Errors in a Program

The symbolic debug program, **sdb** helps find logic errors in a C language program. Refer to Chapter 8, "Debugging Programs" on page 8-1 for information about using this program.

In addition, string searching programs such as **grep**, **sed** and **awk** help locate and change character strings (such as parameter names and syntax problems) in program files. Refer to Chapter 11, "Finding and Changing Strings" on page 11-1 for information about using these programs.

Building and Maintaining a Program

Two programs help control changes to a program and build the final program module. These programs are:

make A program that builds programs from several source modules and that compiles only those modules that have changed. Refer to "Building Programs with make" on page 2-22 for information about using this program.

sccs A program that maintains separate versions of a program without storing separate copies of each version. Refer to Chapter 10, "Maintaining Different Versions of a Program" on page 10-1 for information about using this program.

Shared Libraries

With traditional libraries, each program has a private copy of the program text for each library module it uses. The AIX shared library facility lets multiple processes share the same copy of program text. The text is the executable portion of a program. The programs are linked as before, but the text has been removed from each object module and combined into one text image that is loaded at run time and shared by many programs.

Shared libraries offer the following benefits:

- Programs that use shared libraries may use less disk space because the text sections for shared routines are not linked into each executable program.
- Processes that use shared libraries may require less main memory because the shared text sections are only mapped into main memory once.
- The system may require less time to load programs that use shared libraries because the shared text image may already be in main memory.
- Fewer page faults may be generated when shared libraries are used because the shared text image may already be in main memory. This can result in reduced disk activity which in turn can sometimes result in better response time.

Creating a Shared Library

Use the **shlib** command to create a shared library. **shlib** accepts the names of object modules or archive libraries (created by the **ar** command) as input and creates a shared library as output. See *AIX Operating System Commands Reference* for specific information about using the **shlib** command.

You can use object modules produced by the **C** and FORTRAN compilers without modification as input to **shlib**. You can also use assembler language routines as input if there are no external references in the text section other than **call** or **callr** instructions. You cannot use an object file that is already shared as input.

The **shlib** command copies the text section of each input module into one shared library text image. A shared library ***key*** is added to the header of each input module and the key identifies the name of the shared library text image so each module can be linked correctly by the **ld** command.

It is important that you keep the original object files so you can rebuild the libraries if they change.

Shared Library Keys

You can use the **-k** option of the **shlib** command to assign a specific shared library key. The default key is the name of the first archive or object file on the **shlib** command line. If you do not supply a suffix for the key, the Julian date is appended in the form **yyddd** where **yy** is the year and **ddd** is the sequential day within the year. **shlib** puts the key in each modified output file and **ld** (the link program) includes the key in every program that uses the shared library text image.

You can use the **-r** option of **shlib** to include an arbitrary character string in the shared library text image in addition to the key name.

You can use the **what** command to display all shared library keys a program uses and the key embedded in a shared library text image. Use the **-b** option of the **dump** command to display the key for the shared library an object module belongs to.

Normally, the system appends the key name to each **LIBPATH** directory to form the path name of a possible shared library, in the same way the shell searches for programs using **PATH**. However, if a shared library key begins with a / (slash), it is treated as an absolute name; the system does not append any directories to the key name, and the system attempts to open a file with that name.

Installing Shared Libraries

After you build a shared library, you need to install the text image in an appropriate library directory. If the shared library is not installed in **/lib** or **/usr/lib**, users of programs that are linked to the shared library need to define the **LIBPATH** environment variable to include the directory containing the shared library text image. The **LIBPATH** variable defines the search path for a shared library text image in the same manner that the **PATH** variable defines the search path for a file. The system searches for the shared library as follows:

- If the **LIBPATH** variable is defined, the system combines the path names found in the **LIBPATH** variable and the library key found in the **a.out** header and searches for a shared library with that combined name. If the search does not produce a match, an error message is printed and the program is not started.
- If the **LIBPATH** variable is not defined, then the directories **/lib** and **/usr/lib** are searched for the shared library name.
- If the program has the **setuid** or **setgid** flags set, only **/lib** and **/usr/lib** are searched. This prevents someone from replacing a library call with another, assuming write permission is restricted for the **/lib** and **/usr/lib** directories.

IBM RT Advanced Interactive Executive Operating System Version 2.2

AIX Operating System Programming Tools and Interfaces



SC23-2016-1

When using a shared library with Distributed Services, be aware that any of the following conditions can exist when the program runs:

Location of Main Program	Location of Library
Local node	Local node
Local node	Remote node
Remote node	Local node
Remote node	Remote node

Ensure that the **LIBPATH** variable includes a path to the library being used. For example, when executing a program on a remote node that was compiled with a shared library on the remote node, you can ensure that the program will find the library in one of two ways:

1. Make a local copy of the library in the directory defined by your **LIBPATH** variable.
2. Mount the remote library and add the path to that mounted library to your **LIBPATH** variable.

Linking Shared Libraries

To link (**ld**) shared modules, specify object file names or an archive library name just as you would if you were linking modules that are not shared. When the **ld** command processes shared modules, it gets the key (generated by **shlib**) from the object file or archive library header and places it in the **a.out** file so **exec** can determine what shared library text images are needed at run time. There is an option for the **ld** command to control which segment the text image is mapped into.

It is best to keep the number of text images that must be mapped by a program to a minimum. This could reduce program startup overhead, assuming all the text images are not already mapped by some other process.

If the shared library text image key is different from the file name of the text image, you must use the **-k** option of **ld** and specify the correct file name. See *AIX Operating System Commands Reference* for more information about the **ld** command.

Changing Shared Libraries

Very limited changes are allowed to programs that are a part of a shared library without requiring programs that use the library to relink it:

- You can change integer constant values.
- You cannot change:
 - Floating-point constants
 - String constants.
- You cannot add, remove or rearrange:
 - Static variable definitions
 - External variable definitions
 - References to functions
 - References to external variables.

You can use the **-a** or **-v** option on the **shlib** command to determine whether or not you have to relink your programs after changing a shared library. Both options create a new shared library text image, but instead of building new output files, **shlib** verifies whether or not the new files would be identical to the files created previously. If the output files would be identical, the key of the new shared library text image is identical to the previous key. Verifying that the new output files would be identical is sufficient proof that the existing programs do not have to be relinked. **shlib** reports any discrepancy between the previous key and the new key.

If you make changes to a shared library that cause a change in the output files, you can handle the changes in either of two ways:

- You can build a new version of the shared library and relink the programs that use it.
- You can modify individual archive members locally and continue to use the old shared library.

If you build a new shared library because of program changes and you specify the same key as an existing shared library, programs that use the new shared library may not run correctly until they are relinked because they may still be using the old text image.

To make local changes to an archive that refers to a shared library text image, replace any member of the archive with an individual object file that has not been processed by **shlib**. You should also change the unshared version of the archive so you can generate a new shared library at some later date. When a program is linked with the modified archive, the new member is not shared and the text associated with the member is copied into the program.

You can also replace a member of an archive with a member that refers to a different shared library if the new member has been processed by **shlib**.

Programming Interfaces

When writing an application program for the RT system, use the following system services:

- Shell commands
- Library routines
- System calls.

These services are available from a C language program.

Shell Commands

To include the functions of any of the shell commands in a program, use the **fork** and **exec** system calls to allow the command to run in a part of the system (called a ***process***) that is separate from the program. The **system** library routine also runs a shell command in a program, and the **popen** library routine uses shell filters. When using shell commands in a program, ensure that these commands are also available on all systems that will use the program. Refer to *AIX Operating System Commands Reference* for details about shell commands.

Library Routines

Routines from the system libraries handle many complex or repetitive programming situations so that you can concentrate programming effort on the unique programming situations. Details of each library subroutine are in *AIX Operating System Technical Reference*. Some of the libraries on the system are:

C library
A collection of input/output formatting routines, system call interface routines and other functions. This library includes the library **stdio**, which is the standard buffered input/output system.

Run Time Services library
A collection of routines that help a program use the following system services:

- Configuration
- Messages
- Trace
- Error log.

Math library
A collection of mathematics functions.

extended curses library
A collection of routines for writing programs that help control display screen input and output without regard to the type of terminal that the system uses.

See Chapter 3, "Using the Subroutine Libraries" on page 3-1 for a summary of the functions available in some of the libraries. See Chapter 5, "Controlling the Terminal Screen" on page 5-1 for a description of using the extended curses library.

System Calls

System calls offer some additional control of the system, allow more detailed control of input and output operations, and provide functions not available through the library routines. However, system calls do not format the data, or take care of system housekeeping like the library routines do. Therefore, when using system calls, be sure to provide data stream control and housekeeping functions. Each system call is explained in *AIX Operating System Technical Reference*. See Chapter 4, "Using System Calls" on page 4-1 for examples of how to use many of the system calls.

Using the Programming Examples

Note: The Programming Examples in this book have been tested using Version 2.1.1 of the operating system software. Changes that may be made to subsequent releases of the operating system may change how these programs operate. Be sure that the level of the Programming Examples is compatible with the level of the operating system at your site.

This book uses sample programs to demonstrate the tools and interfaces in the system. These programs are working models that perform a task to demonstrate the function that the text explains. The source code (in C language) for these programs is on the diskette located in the back of this book. To install the Programming Examples, refer to Appendix A, " Installing Programming Examples" on page A-1.

Although the programs may not perform useful functions, use the syntax and declarations in the sample programs as templates for program development. Look at or change the programs in these files as needed. However, if you change the program, change only a copy of it. Please keep the original sample program files so that another person using this book can also use them.

The sample program files contain detailed comments that explain how the program operates. This book includes many of the program listings, but because of space restrictions, the comments are not printed. To print the listings of the program files, use the command:

```
print filename
```

Where *filename* is the path name of the file to print. For example, to print the file that contains the source program for the signal system call (used in Chapter 4, "Using System Calls" on page 4-1), use the following command:

```
print /usr/lib/samples/toolbook/sigtst.c
```

You should also compile and run these programs. Compiling the programs helps you become familiar with the compiler command **cc**. Seeing the program run (most of the sample programs display output to the screen) helps you understand the idea that the sample program demonstrates. To compile the programs, copy the programs from the directory **/usr/lib/samples/toolbook** to your current directory:

```
cp /usr/lib/samples/toolbook/* .
```

To compile a program from that directory into a file that can be run, use the command:

```
cc filename.c -o filename
```

where *filename* is the name of the file to compile. For example:

```
cc sigtst.c -o sigtst
```

compiles the signal sample program and puts the output in a file named **sigtst**. Run this file by entering the command:

```
sigtst
```

If problems occur with the program, compare it with the listing in this book. Except for the comments, the program should be identical to the listing in this book.

Chapter 2. Compiling and Linking Programs

CONTENTS

About This Chapter 2-2
Compiling a Program 2-3
 Choosing a Compiler 2-3
 Using the cc Program 2-4
Checking C Programs 2-5
 Operation 2-6
 Program Flow 2-6
 Data Type Checking 2-7
 Variable and Function Checking 2-10
 Using Variables Before They Are Initialized 2-12
 Portability Checking 2-12
 Coding Errors and Style Differences 2-14
 Creating a lint Library 2-15
Other C Programming Tools 2-19
Processing Assembler Language Routines 2-20
 Using the as Program 2-20
 Using the ld Program 2-21
 Using the cc Program 2-21
Building Programs with make 2-22
 Operation 2-22
 Using the make Program 2-23
 Description Files 2-24
 Internal Rules 2-29
 Defining Default Conditions 2-35
 Including Other Files 2-35
 Defining Macros 2-36
 Using Macros in a Description File 2-37
 Internal Macros 2-38
 Changing Macro Definitions in a Command 2-42
 Using make with SCCS Files 2-43
 How make Uses the Environment Variables 2-44
 Example of a Description File 2-45

About This Chapter

To make source code into a program that the system can run, process the source file with a compiler program and a linker program. The compiler changes the source statements into object code that the computer can run; the linker connects program modules together and determines how to put the finished program into memory. This chapter discusses the following programming processes:

- Compiling the program
- Checking C programs
- Other C programming tools
- Processing assembler language routines
- Linking the program
- Building the program using the **make** utility program.

This chapter does not contain complete information about any of the programs. For complete information refer to the reference book for the language compiler or to *AIX Operating System Commands Reference* (for the C language compiler).

Compiling a Program

A ***compiler*** is a program that reads program text from a file and changes the programming language statements in that file to a form that the system can understand. The following steps show how the system creates this final form of the program:

1. Includes additional files specified with the `#include` directive, and expands macros into programming language statements.
2. Changes the programming language statements into assembler language code using the language compiler.
3. Changes the assembler language into object code (a form that the system can understand) using the assembler (**as** command). The object code is stored in a file with a **.o** suffix. This form of the program cannot be executed.
4. Links the object code (using the **ld** command) into a program that the system can execute. If you do not specify differently, the executable program is in the file **a.out** in the current directory.

If the program is written in the C language, use the **cc** program to perform these steps. See "Using the cc Program" on page 2-4 and *AIX Operating System Commands Reference* for information about this program. If the program is written in assembler language, see "Processing Assembler Language Routines" on page 2-20.

Choosing a Compiler

The following list includes some of the programming languages that are available for use with the AIX Operating System:

- Basic
- FORTRAN
- Pascal
- C
- IBM RT Assembler Language.

The books that come with the compiler programs contain information for using those languages. The examples in this chapter use the C language.

You can also write parts of the program in different languages and have one main routine call the separate routines to execute. To do this, however, follow the rules explained in *Assembler Language Reference* under the topic of ***calling conventions***.

Using the cc Program

The **cc** program calls the C language compiler, but it can do much more. The **cc** program can:

- Process the input with a macro preprocessor
- Compile a high-level language program
- Assemble an assembler language program
- Link program modules.

You can select any or all of these functions. In addition, you can replace the supplied programs for any of these steps with a program suited to special needs. *AIX Operating System Commands Reference* contains detailed reference information about the **cc** program.

Examples of Commands

The following examples show some operations with the **cc** program using command line flags.

Compile source file `testfile.c` using the C library (**libc.a**) and the run-time library (**librts.a**), link the resulting module and place the output in **a.out**:

```
cc testfile.c
```

Process source file `testfile.c` to produce assembler language output, and place the output in `testfile.s`:

```
cc testfile.c -S
```

Compile source file `testfile.c` using the C library (**libc.a**) and the run-time library (**librts.a**), and place the unlinked output in `testfile.o`:

```
cc testfile.c -c
```

Process source file `testfile.c` using the macro preprocessor only, and place the output in `testfile.i`:

```
cc testfile.c -P
```

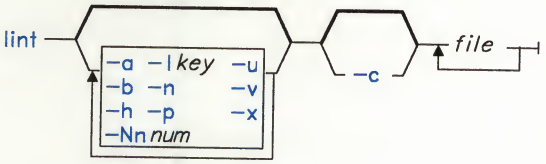
Compile source file `testfile.c` using the C library (**libc.a**) and the run-time library (**librts.a**), but using the `newcpp` and `newcomp` compiler programs in the directory `/u/jim`. Place the unlinked output in `testfile.o`:

```
cc testfile.c -c -B/u/jim/new -tp0
```

Checking C Programs

Use the **lint** program to ensure that C programs do not contain syntax errors, as well as to verify that the programs do not contain data type errors. The **lint** program checks these areas of a program more carefully than the C compiler does, and displays many messages that point out possible problems. These messages may not require you to change the program if you decide to ignore the possible problems.

To start **lint**, enter the command according to the following diagram as shown in *AIX Operating System Commands Reference*:



A5AC6001

The parameters for the **lint** command are in the following categories:

flags Optional flags to control **lint** messages. This section contains examples of some useful flags. See *AIX Operating System Commands Reference* for a complete list of the flags for the system.

filename The name of the C language source file for **lint** to check. The file name must end with **.c**.

library-name The name of a library that **lint** uses when checking the program. The following libraries are included with the system:

-ldos	Checks DOS file library call syntax
-lcurses	Checks extended curses library call syntax
-m	Checks math library call syntax
-port	Checks for portability with other systems.

You can also create your own lint library. See "Creating a lint Library" on page 2-15 for more information.

With no flags specified on the command line, the **lint** program checks the C source files and writes messages about the following coding errors and programming style differences that it finds:

- Data types that are not used correctly
- Variables and functions that are not used
- Functions that are not used correctly
- Syntax errors
- Techniques that could cause problems in moving the program to other systems.

Operation

The **lint** program checks a group of files using the following procedure:

1. Checks each file and writes messages for problems found in that file
2. Collects errors in included files and writes those messages
3. Checks for consistency of labels and data types among the group of files
4. Writes the source file name followed by a ? (question mark) if any errors remain that are not assigned to either a source file or an included file.

If **lint** does not report any errors, the program has correct syntax and will compile without errors. Passing that test, however, does not mean that the program will operate correctly, or that the logic design of the program is accurate. The **lint** program does not check for design problems. It only checks language semantics and syntax.

Program Flow

The **lint** program detects parts of the program that cannot be reached. It writes messages about statements that do not have a label, but immediately follow statements that change the program flow, such as:

- goto
- break
- continue
- return.

The **lint** program also detects and writes messages for the following conditions:

- A loop that cannot be exited at the bottom
- A loop that cannot be entered at the top
- Infinite loops such as:
 - while(1)
 - for(;;)

Some programs that work may have such loops. However, the loops can cause problems.

The **lint** program does not detect functions that are called, but never return to the calling program. For example, a call to **exit** may result in code that cannot be reached, but **lint** does not detect it.

Programs generated by **yacc** and **lex** may have hundreds of **break** statements that cannot be reached. The **lint** program normally writes an error message for each of these **break** statements. Use the **-O** flag for the **cc** command when compiling the program to eliminate the resulting object code inefficiency, so that these extra statements are not important. Use the **-b** flag with the **lint** program to prevent writing of these messages when checking **yacc** and **lex** output code.

Data Type Checking

The **lint** program enforces the type checking rules of C language more strictly than the compiler does. In addition to the checks that the compiler makes, **lint** checks for the data type errors in the following areas:

- Binary operators and implied assignments
- Structures and unions
- Function definition and uses
- Enumerators
- Type checking control
- Type casts.

Binary Operators and Implied Assignments

The C language allows mixing of the following data types in statements, and the compiler does not indicate an error when they are mixed:

- char
- short
- int
- long
- unsigned
- float
- double.

The language converts data types within this group automatically to allow the programmer more flexibility in programming. This flexibility, however, means that the programmer, *not* the language, must ensure that the data type mixing produces the desired result.

You can mix these data types when using them in the following ways (in the examples, `alpha` is type **char**, and `num` is type **int**):

- Operands on both sides of an assignment operator, for example:

```
alpha = num;
```

- Operands in a conditional expression, for example:

```
value = ( alpha < num ) ? alpha : num;
```

- Operands on both sides of a relational operator, for example:

```
if( alpha != num )
```

- The type of an argument in a **return** statement is converted to the type of the value that the function returns. For example:

```
funct(x)        /* returns an integer */
{
  return( alpha );
}
```

The data types of pointers must agree exactly, except that you can mix arrays of `x`'s with pointers to `x`'s.

Structures and Unions

The **lint** program checks structure operations for the following requirements:

- The left operand of the -> operator must be a pointer to a structure.
- The left operand of the . operator must be a structure.
- The right operand of these operators must be a member of the same structure.

The **lint** program makes similar checks for references to unions.

Function Definition and Uses

The **lint** program applies strict rules to function argument and return value matching. Arguments and return values must agree in type with the following exceptions:

- You can match arguments of type **float** with arguments of type **double**.
- You can match arguments within the following types:
 - char
 - short
 - int
 - unsigned.
- You can match pointers with the associated arrays.

Enumerators

The **lint** program checks enumerated data type variables to ensure that:

- Enumerator variables or members are not mixed with other types or other enumerators
- The enumerated data type variables are only used in the following areas:
 - Assignment (=)
 - Initialization
 - Equivalence (==)
 - Not equivalence (!=)
 - Function arguments
 - Return values.

Type Checking Control

To turn off strict type checking for one expression in the program, add the directive:

```
/*NOSTRICT*/
```

to the program immediately before the expression. This directive prevents strict type checking for only the next line in the program.

Type Casts

Type casts in the C language allows the program to treat data of one type as if it were data of another type. The **lint** program can check for type casts and write a message if it finds one.

The **-c** flag for the **lint** program controls the writing of comments about casts. Without the **-c** flag, **lint** treats casts as though they were assignments subject to messages. The resulting messages indicate the casts that are in the program. With the **-c** flag, **lint** ignores all legal casts.

Variable and Function Checking

The **lint** program detects variables and functions declared in the program, but not used. When it finds one of these cases, it writes a message. Variable and function errors that **lint** finds include the following:

- Functions that return values inconsistently
- Variables and functions that are defined, but not used
- Arguments to a function call that are not used
- Functions that can return either with or without values
- Functions that return values that are never used
- Programs that use the value of a function when the function does not return a value.

Inconsistent Function Return

If a function returns a value under one set of conditions, but does not return a value under another set of conditions, you cannot predict the results of the program. The **lint** program detects this type of error. For example, if both of the following statements are in a function definition:

```
return( expr );
```

and

```
return;
```

The **lint** program writes the message:

```
function name contains return(e) and return
```

When using this function, the program may or may not receive a return value. The error message points out that problem.

The **lint** program also detects function returns caused by reaching the end of the function code (an implied return). For example, in the following part of a function:

```
checkout (a)
{
        if (a) return (3);
        fix_it ();
}
```

If a tests false, `checkout` calls `fix_it` and then returns with no defined return value. In this case, **lint** writes the message:

```
function checkout contains return(e) and return
```

If `fix_it`, like `exit`, never returns, **lint** still writes the message even though nothing is wrong.

Function Values That Are Not Used

The **lint** program detects cases where a function returns a value and the calling program may not use the value. If the value is never used, the function definition may be inefficient and should be checked. If the value is sometimes used, the function may be returning an error code that the calling program does not check.

Disabling Function Related Error Messages

To prevent **lint** from reporting these types of errors, specify one or more of the following flags to the **lint** command:

-x Do not write messages about variables that are declared in an **extern** statement, but are never used.

-v Do not write messages about arguments to functions that are not used (except those that are also declared as register arguments).

-u Do not write messages about functions and external variables that are either used and not defined, or defined and not used. Use this flag to run **lint** on a subset of files of a larger program.

To prevent **lint** from reporting errors for one function add the directive:

```
/*ARGSUSED*/
```

to the program before the function.

Add the following directive before the function definition to prevent the program from writing messages about variable numbers of arguments in calls to a function:

```
/*VARARGSn*/
```

To check the first several arguments and leave the later arguments unchecked, add a digit to the end of the VARARGS directive to give the number of arguments that should be checked, such as:

```
/*VARARGS2*/
```

When **lint** reads this directive, it checks only the first two arguments.

When using **lint** with some (but not all) files that operate together, many of the functions and variables defined in those files may not be used. Also, many functions and variables defined elsewhere may be used. Use the **-u** flag to prevent **lint** from writing these messages.

Using Variables Before They Are Initialized

The **lint** program detects if a program uses a local variable (automatic and register storage classes) before assigning a value to it. In this case, using a variable also includes taking the address of the variable. This is because the program can use the variable (through its address) any time after it knows the address of the variable. Therefore, if the program does not assign a value to the variable before it finds the address of the variable, **lint** reports an error. Because **lint** only checks the physical order of the variables and their usage in the file, it may write messages about a program that actually does not contain errors.

The **lint** program recognizes and writes messages about:

- Initialized automatic variables
- Variables that are used in the expression that first sets them
- Local variables that are set and never used.

Note: The operating system initializes static and external variables to zero. Therefore, **lint** assumes that these variables are set (to zero) at the start of the program, and does not check to see if they have been assigned a value when they are used. When developing a program for a system that does not do this initialization, ensure that the program sets static and external variables to an initial value.

Portability Checking

Use **lint** to help ensure that you can compile and run the program on other systems that have a C language compiler that conforms to the UNIX[1] System V requirements for a C compiler. The following paragraphs indicate areas to check before compiling the program

1 UNIX was developed and licensed by AT&T. It is a registered trademark of AT&T in the United States of America and other countries.

on another system. Checking only these areas, however, does not guarantee that the program will run on any system.

Character Uses

Some systems define characters in a C language program as signed quantities with a range from -128 to 127; other systems define characters as positive values. The **lint** program writes messages when it finds character comparisons or assignments. The messages indicate that the use of characters may not be portable to other systems. For example, the fragment:

```
char c;
.
.
.
if( ( c = getchar() ) <0 ) . . .
```

may work on one system but fail on systems where characters always take on positive values. The **lint** program writes the message:

```
nonportable character comparison
```

when it checks the program.

To make the program work on systems that use positive values for characters, declare `c` as an integer because **getchar** returns integer values.

Bit Field Uses

Bit fields may also produce problems when transferring a program to another system. When assigning constant values to bit fields, the field may be too small to hold the value, because bit fields may be signed quantities on the new system. To make this assignment work on all systems, declare the bit field to be of type **unsigned** before assigning values to it.

External Name Size

When changing from one type of system to another, be aware of differences in the information retained about external names during the loading process. The number of characters allowed for external names can vary. The AIX Operating System C language compiler considers at least the first 64 characters in internal and external identifiers as significant. Some programs that the compiler command calls and some of the functions that your programs call may further limit the number of significant characters in identifiers. In addition, the compiler adds a leading underscore to all names, and keeps uppercase and lowercase characters separate. On other systems, uppercase or lowercase may not be important or allowed. To avoid problems with loading the program when transferring from one system to another:

1. Find out the requirements of each system
2. Run **lint** with the **-p** flag.

The **-p** flag tells **lint** to change all external symbols to one case and limit them to six characters while checking the input files. The messages produced indicate the terms that may need to be changed.

Multiple Uses and Side Effects

Be careful when using complicated expressions. Many C compilers evaluate complex expressions in different orders. Function calls that are arguments of other functions may or may not be treated the same as ordinary arguments. Also, operators such as assignment, increment, and decrement may cause problems when used on another system. For example, if any variable is changed by a side effect of one of the operators and is also used elsewhere in the same expression, the result is undefined. For example, the evaluation of the variable `years` in the following example is confusing because on some machines `years` is incremented before the function call, while on other machines `years` is incremented after the function call:

```
printf( "%d %d\n", ++years, amort( interest, years ) );
```

The **lint** program checks for simple scalar variables that may be affected by evaluation order problems. For example, the statement:

```
a[i]=b[i++];
```

causes **lint** to write the message:

```
warning: i evaluation order undefined
```

Coding Errors and Style Differences

Use **lint** to detect some coding errors and differences in coding style from the style that **lint** expects. Although coding style is mainly a matter of individual taste, examine each difference to ensure that the difference is both needed and accurate. The following paragraphs indicate the types of coding and style problems that **lint** can find.

Assignments of Long Variables to Integer Variables

If you assign variables of type **long** to variables of type **int**, the program may not work properly. The long variable is truncated to fit in the integer space and data may be lost. An error of this type occurs frequently when converting a program that uses **typedefs** to run on a different system. When changing a **typedef** variable from **int** to **long**, the program can stop working because an intermediate result may be assigned to an integer variable, and the intermediate result is truncated.

To assign a long variable to an integer variable and prevent **lint** from writing messages for these assignments, use the **-a** flag with the **lint** program.

Operator Precedence

The **lint** program detects errors in operator precedence. Without parentheses to show order in complex sequences, these errors are hard to find by looking at the code. For example, the following statements are not clear:

```
if(x&077==0) . . .          /* actually: if(x & (077 == 0) ) */
                            /* should be: if( (x & 077) == 0) */
```

or

```
x<<2+40                     /* shift x left 42 positions */
                            /* should be: (x<<2) + 40 */
```

Use parentheses to make the operation more clearly understood. If you do not, **lint** writes a message.

Conflicting Declarations

The **lint** program writes messages about variables that are declared in inner blocks in a way that conflicts with their use in outer blocks. This practice is allowed but may cause problems in the program. Use the **-h** flag with the **lint** program to prevent writing of messages about conflicting declarations.

Creating a lint Library

For programming projects that define additional library routines, create an additional lint library to check the syntax of the programs. Using this library, the **lint** program can check the new functions in addition to the standard C language functions. Perform the following steps to create a new lint library (see the following paragraphs for more information about these steps).

1. Create an input file that defines the new functions
2. Process the input file to create the lint library file
3. Run **lint** using the new library.

Creating the Input File

Figure 2-1 on page 2-17 shows an input file that defines three additional functions for **lint** to check. This file is a text file that you create with an editor. It consists of:

1. A directive to tell the **cpp** program that the following information is to be made into a library of **lint** definitions:

   ```
   /*LINTLIBRARY*/
   ```

2. A series of function definitions that define:

 - The type of the function (**int** in the example)
 - The name of the function
 - The parameters that the function expects
 - The types of the parameters
 - The value that the function returns.

Name this file in the following format:

```
llib-l pgm
```

In this format, the letters *pgm* represent a unique name that indicates the functions contained in the input file. For example, in the example input file the name of this input file could be **llib-ldms**. When choosing the name of the file, ensure that it is not the same as any of the existing files in the **/usr/lib** directory.

```
/*LINTLIBRARY*/

#include <dms.h>


int  dmsadd( rmsdes, recbuf, reclen )
                int rmsdes;
                char *recbuf;
                unsigned reclen;
              { return 0; }
int  dmsclos( rmsdes)
                int rmsdes;
              { return 0; }
int  dmscrea( path, mode, recfm, reclen )
                char *path;
                int mode;
                int recfm;
                unsigned reclen;
              { return 0; }
```

Figure 2-1. Example lint Library Input File

Creating the lint Library File

To create a lint library file, process the input file using the following command:

```
$ /lib/cpp -C -Dlint llib-lpgm | /usr/lib/lint1 -Htmpfile > \
  /usr/lib/llib-lpgm.ln
```

This command tells the preprocessor program **cpp** and an intermediate program **lint1** to create a lint library file, `/usr/lib/llib-lpgm.ln` using the input file `llib-lpgm`. In each of these cases, the *pgm* in the file name represents the identifier for the input file. The file name `tmpfile` can be any temporary file name. The **lint1** program creates this file and uses it for intermediate storage. When the program completes, delete this file:

```
rm tmpfile
```

Checking a Program with the New Library

To check a program using the new library, use the command:

```
lint -lpgm filename.c
```

In this command, the letters *pgm* represent the identifier for the library, and *filename.c* represents the name of the file containing the C language source code to check. With no other flags, the **lint** program checks the C language source code against the standard lint library in addition to checking the indicated special lint library.

Other C Programming Tools

The AIX Operating System provides tools to help format and check the structure of the C language program. These tools include:

cb ***c beautifier***: This program formats the C language source program into a form that uses indention levels to show the structure of the program.

cflow ***c flow diagram generator***: This program produces an output diagram that shows the logic flow of the C language source program.

cxref ***c cross reference list***: This program produces a list of all external references for each module of the C language program, including where the reference is resolved (if it is).

AIX Operating System Commands Reference explains the syntax and options for these programs. The sample programs contain a C language source program that shows the effects of the **cb** command. This file, `cbtest`, contains the program from one of the other sample programs used in this book. However, the program in this file is without format. Use the **pg** command to look at this file:

```
pg cbtest
```

Press **Enter** at the end of each screen. Then use the **cb** command to format the file and put the formatted output in a file, `test.pretty.c`:

```
cb cbtest > test.pretty.c
```

Look at the output file `test.pretty.c` using the **l** command to see the effect of the **cb** command.

Processing Assembler Language Routines

To use program modules written in RT assembler language in a program, assemble the source code and link the resulting output with any other modules in the program. To perform these steps, either:

1. Use the **as** program to assemble the source code into an object module.
2. Use the **ld** program to link the object modules with the other object modules that form the program.

or

1. Use the **cc** program to both assemble and link the program.

Using the as Program

The following command sequences show some uses of the **as** program to assemble an assembler language module into an object module:

- Assemble source file `asmtest.s` and place the output in the default file, **a.out**.

```
as asmtest.s
```

- Assemble source file `asmtest.s` and place the output in the file `myfile.o`.

```
as -o myfile.o asmtest.s
```

The following command sequence shows how to use the **as** program to generate an assembler listing:

- Assemble source file `asmtest.s` and place the assembler listing in the file, `asmtest.lst`.

```
as -l asmtest.s
```

- Assemble source file `asmtest.s` and place the assembler listing in the file `myfile`.

```
as -lmyfile asmtest.s
```

For more information about the **as** command, see **as** in *AIX Operating System Commands Reference*.

Using the ld Program

After assembling the source program with the **as** program, use the **ld** program to link that object module with other object modules, or to prepare it to run on the system.

Using the cc Program

To use the **cc** program to process an assembler language file, the file name must end in **.s** to indicate that it is an assembler language source file. The following command sequences show some uses of the **cc** program to assemble an assembler language module into an object module, and link it with other object modules to form the program:

- Assemble and link the file `asmtest.s` and place the resulting program in file **a.out**.

  ```
  cc asmtest.s
  ```

- Assemble the file `asmtest.s` and place the resulting unlinked object code in file `asmtest.o`.

  ```
  cc asmtest.s  -c
  ```

- Assemble the file `asmtest.s`, link it with object files `oldfile.o` and `otherfile.o`, and place the resulting program in file **a.out**.

  ```
  cc asmtest.s oldfile.o otherfile.o
  ```

You can also use the **cc** program to generate an assembler listing.

- Assemble the file `asmtest` and place the assembler listing in `asmtest.c`.

  ```
  cc -X asmtest.s
  ```

Building Programs with make

The **make** program builds up-to-date versions of programs. It keeps track of the commands that are needed to create the files, and uses a list of files that must be current before the operations can be done. After changing any part of a program, enter the **make** command on the command line. The **make** program then creates only the files that are affected by the change, according to the rules in its rules file.

Using the **make** program to maintain programs, you can:

- Combine the instructions for creating a large program in a single file
- Define macros to use within the **make** description file
- Define new flags to use with the **make** program
- Create any file to use with the operating system, including SCCS files
- Use shell commands to define the method of file creation, or use the **make** program to create many of the basic types of files
- Create libraries
- Include files from other programs when creating a file.

The **make** program is most useful for medium-sized programming projects. It does not solve the problems of maintaining more than one source version and describing huge programs (see Chapter 10, "Maintaining Different Versions of a Program" on page 10-1).

Operation

Note: The **make** program depends on time stamps on files. When using **make** on a system with Distributed Services installed, ensure that the system clocks show the same date and time for all nodes that contain files for **make** to process.

The **make** program uses the following sources of information:

- A description file that you create
- File names
- Time stamps of the files from the file system
- Rules in the **make** program that tell how to build many of the standard types of files.

The file containing the completed program is called a **target file**. The **make** program creates a target file using a step-by-step procedure:

1. Finds the name of the target file in the description file, or in the **make** command
2. Ensures that the files on which the target file depends exist and are up-to-date

3. Determines if the target file is up-to-date with the files it depends on
4. If the target file or one of the parent files is out of date, creates the target file using one of the following:
 a. Commands from the description file
 b. Internal rules to create the file (if they apply)
 c. Default rules from the description file.

If all files in the procedure are up-to-date when running the **make** program, **make** displays a message to indicate that the file is up-to-date, and then stops. If some files have changed, **make** creates only those files that are out of date, and does not create files that are already current.

When the **make** program runs commands to create a target file, it replaces macros with their values, writes each command line, and then passes the command to a new copy of the shell.

Using the make Program

Start the **make** program from the directory that contains the description file for the file to create. The variable name *desc-file* represents the name of that description file. Then, enter the command:

```
make -f desc-file
```

on the command line. Enter macro definitions, flags, description file names, and target file names along with the **make** command on the command line as follows:

```
make [flags] [macro definitions] [targets]
```

The **make** program then examines the command line entries to determine what to do. First, it looks at all macro definitions on the command line (entries that are enclosed in quotes and have equal signs in them) and assigns values to them. If it finds a definition for a macro on the command line different from the definition for that macro in the description file, it chooses the command line definition for the macro.

Next, the **make** program looks at the flags. See *IBM RT AIX Operating System Commands Reference* for a list of the flags that **make** recognizes.

The **make** program expects the remaining command line entries to be the names of target files to be created. The **make** program creates the target files in left to right order. Without a target file name, the **make** program creates the first target file named in the description file that does not begin with a period. With more than one description file specified, **make** searches the first description file for the name of the target file.

Description Files

The description file tells **make** how to build the target file, what files are involved, and what their relationships are to the other files in the procedure. The description file contains the following information:

- Target file name
- Parent file names that make up the target file
- Commands that create the target file from the parent files
- Definitions of macros in the description file.

The **make** program determines what files to create to get an up-to-date copy of the target file by checking the dates of the parent files. If any parent file was changed more recently than the target file, **make** creates the files that are affected by the change, including the target file.

If you name the description file **makefile** or **Makefile**, and are working in the directory containing that description file, enter the command:

```
make
```

to bring the first target file and its parent files up-to-date, regardless of the number of files that were changed since the last time **make** created the target file. In most cases, the description file is easy to write and does not change often.

To keep many different description files in the same directory, name them differently. Then, enter the command:

```
make -f desc-file
```

substituting the name of the description file to use in place of the variable name *desc-file*.

Format of a Description File Entry

The general form of an entry is:

```
target1 [target2..]:[:] [parent1..][; commands] [#..]
[(tab) commands] [# . . . ]
 . . .
```

The items that are inside brackets are optional. Targets and parents are file names (strings of letters, numbers, periods, and slashes). **make** recognizes wildcard characters such as * (asterisk) and ? (question mark). Each line in the description file that contains a target file name is called a dependency line. Lines that contain commands must begin with a tab character.

Note: Because **make** uses the dollar sign symbol ($) to designate a macro, do not use that symbol in file names of targets and parents, or in commands in the description file unless you are using a defined **make** macro.

Put comments in the description file by using a # (number sign) to begin the comment phrase. The **make** program ignores the # and all characters on the same line after the #. The **make** program also ignores blank lines.

If the line is not a comment line, you can enter lines that are longer than the line width of the input device. To continue a line on the next line, put a \ (backslash) at the end of the line that is to be continued.

Using Commands in a Description File

A command is any string of characters not including a # or a new line. A command can use a # if it is in quotes. Commands can appear either after a semicolon on a dependency line, or on lines beginning with a tab immediately following a dependency line.

When defining the command sequence for a particular target, specify either one command sequence for each target in the description file, or specify separate command sequences for special sets of dependencies. Do not do both.

To use one command sequence for every use of the target, use a single : (colon) following the target name on the dependency line. For example:

```
test:       dependency list1...;
        command list...
            .
            .
            .
test:       dependency list2...;
```

defines a target name, `test`, with a set of parent files, and a set of commands to create the file. The target name, `test`, can appear in other places in the description file with another dependency list, but that name cannot have another command list in the description file. When one of the files that `test` depends on changes, **make** runs the commands in that one command list to create the file, `test`.

To specify more than one set of commands to create a particular target file, enter more than one dependency definition. Each dependency line must have the target name, followed by :: (two colons), a dependency list, and a command list that **make** uses if any of the files in the dependency list changes. For example:

```
test::      dependency list1...;
          command list1...

test::      dependency list2...;
          command list2...
```

defines two separate processes to create the target file, `test`. If any of the files in dependency list1 changes, **make** runs command list1; if any of the files in dependency list2 changes, **make** runs command list2. To avoid conflicts, a parent file cannot appear in both dependency list1 and dependency list2.

Note: Because **make** passes the commands from each command line to a new shell, be careful when using certain commands (for example, **cd** and shell control commands) that have meaning only within a single shell process. The **make** program forgets these results before running the commands on the next line.

To group commands together, use the \ (backslash) at the end of a command line. The **make** program continues that command line into the next line in the description file. The shell sends both of these lines to a single new shell.

Calling the make Program from a Description File

Nest calls to the **make** program within a **make** description file by including the **$(MAKE)** macro in one of the command lines in the file. If this macro is present, **make** calls another copy of **make** even if the **-n** flag (*do not execute*) is set. The **make** program passes the flags to the new copy of **make** through the **MAKEFLAGS** variable.

If the **-n** flag is set when the **$(MAKE)** macro is found, the new copy of **make** does not do any of its commands, except another **$(MAKE)** macro. Use this characteristic to test a set of description files that describe a program. Enter the command:

```
make -n
```

The **make** program does not do any of the operations, but it writes all of the steps needed to build the program, including output from lower level calls to **make**.

Preventing the make Program from Writing Commands

To prevent **make** from writing the commands while it runs, do any of the following:

- Use the **-s** flag on the command line when using the **make** command.
- Put the fake target name **.SILENT** on a dependency line by itself in the description file. Because **.SILENT** is not a real target file, it is called a ***fake target***.
- Put an @ in the first character position of each line in the description file that **make** should not write.

Prevent Stopping on Errors

The **make** program normally stops if any program returns an error code that is not zero. Some programs return status that has no meaning.

To prevent **make** from stopping on errors, do any of the following:

- Use the **-i** flag on the command line when using the **make** command.
- Put the fake target name **.IGNORE** on a dependency line by itself in the description file. Because **.IGNORE** is not a real target file, it is called a ***fake target***.
- Put a - (hyphen) in the first character position of each line in the description file where **make** should not stop on errors.

Example of a Description File

For example, a program named **prog** is made by compiling and loading three C language files `x.c`, `y.c`, and `z.c` with the C library (**libc.a**) and the run-time library (**librts.a**). The files `x.c` and `y.c` share some declarations in a file named `defs`. The file `z.c` does not share those declarations. A description file to create **prog** looks like:

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog

# Make x.o from 2 other files
x.o:   x.c defs
# Use the cc program to make x.o
    cc  -c x.c

# Make y.o from 2 other files
y.o:   y.c defs
# Use the cc program to make y.o
    cc  -c y.c

# Make z.o from z.c
z.o:   z.c
# Use the cc program to make z.o
    cc  -c z.c
```

If this file is called **makefile**, just enter the command:

```
make
```

to make **prog** up-to-date after making changes to any of the four source files `x.c`, `y.c`, `z.c` or `defs`.

Making the Description File Simpler

To make this file simpler, use the internal rules of the **make** program. Using file system naming conventions, **make** knows that there are three **.c** files corresponding to the needed **.o** files. It also knows how to generate an object from a source file (that is, issue a **cc -c** command).

By taking advantage of these internal rules, the description file becomes:

```
# Make prog from 3 object files
prog:  x.o y.o z.o
# Use the cc program to make prog
    cc  x.o y.o z.o -o prog

# Use the file defs and the .c file
# when making x.o and y.o
x.o y.o:   defs
```

Internal Rules

The internal rules for the **make** program are in a file that looks like a description file. With the **-r** flag the **make** program does not use the internal rules file; you must supply the rules to create the files. The internal rules file contains a list of file name suffixes (such as, **.o**, or **.a**) that **make** understands, plus the rules that tell **make** how to create a file with one suffix from a file with another suffix. If you do not change the list, **make** understands the following suffixes:

.o Object file

.c C source file

.e Efl source file

.r Ratfor source file

.f FORTRAN source file

.s Assembler source file

.y Yacc-c source grammar

.yr Yacc-Ratfor source grammar

.ye Yacc-Efl source grammar

.l Lex source grammar

The list of suffixes is like a dependency list in a description file, and follows the fake target **.SUFFIXES**. Because **make** looks at the suffixes list in left to right order, the order of the entries is important. The **make** program uses the first entry in the list that satisfies two requirements:

- The entry matches input and output suffix requirements.
- The entry has a rule assigned to it.

The **make** program creates the name of the rule from the two suffixes of the files that the rule defines. For example, the name of the rule to transform a **.r** file to a **.o** file is **.r.o**.

To add more suffixes to the list, add an entry for **.SUFFIXES** in the description file. For a **.SUFFIXES** line without any suffixes following the target name in the description file, **make** erases the current list. To change the order of the names in the list, erase the current list and then assign a new set of values to **SUFFIXES**.

Figure 2-2 shows the paths that **make** uses to create a file. If two paths connect a pair of suffixes, **make** uses the longer one only if the intermediate file exists or is named in the description file.

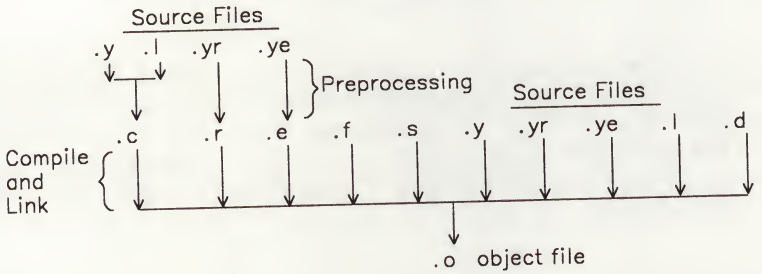


Figure 2-2. Rules for Creating Files

Example of Default Rules File

Figure 2-3 on page 2-31 shows a portion of the default rules file.

```
# Define suffixes that make knows
.SUFFIXES:  .o .c .e .r .f .y .yr .ye .l .s

# Begin macro definitions for
# internal macros
YACC = yacc
YACCR = yacc -r
YACCE = yacc -r
YFLAGS =
LEX = lex
LFLAGS =
CC =cc
AS = as
CFLAGS =
RC = ec
RFLAGS =
EC = ec
EFLAGS =
FFLAGS =
# End macro definitions for
# internal macros

# Create a .o file from a .c
# file with the cc program
.c.o:
        $(CC) $(CFLAGS) -c $<

# Create a .o file from either a
# .e , a .r , or a .f
# file with the efl compiler
.e.o .r.o .f.o:
        $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
```

Figure 2-3 (Part 1 of 2). Example Default Rules File

```
# Create a .o file from
# a .s file with the assembler
.s.o:
        $(AS) -o $@ $<


.y.o:
# Use yacc to create an intermediate file
        $(YACC) $(YFLAGS) $<
# Use cc compiler
        $(CC) $(CFLAGS) -c y.tab.c
# Erase the intermediate file
        rm y.tab.c
# Move to target file
        mv y.tab.o $@

.y.c:
# Use yacc to create an intermediate file
        $(YACC) $(YFLAGS) $<
# Move to target file
        mv y.tab.c $@
```

Figure 2-3 (Part 2 of 2). Example Default Rules File

Single Suffix Rules

The **make** program also has a set of single suffix rules to create files directly to a file name that does not have a suffix (command files for example). The **make** program has rules to change the following source files with a suffix to object files without a suffix:

.c: From a C language source file

.c~: From an SCCS C language source file

.sh: From a shell file

.sh~: From an SCCS shell file

Therefore, to maintain a program like **cat**, enter:

```
make cat
```

if all of the needed files are in the current directory.

Using Make with Archive Libraries

Use **make** to build libraries and library files. The **make** program recognizes the suffix **.a** as a library file. The internal rules for changing source files to library files are:

.c.a C source to archive

.c~.a SCCS C source to archive

.s~.a SCCS assembler source to archive

Changing Macros in the Rules File

The **make** program uses macro definitions in the rules file. To change these macro definitions, enter new definitions for those macros on the command line or in the description file. The **make** program uses the following macro names to represent language processors that it uses:

- AS for the assembler
- CC for the C compiler
- RC for the ratfor compiler
- EC for the efl compiler
- YACC for yacc
- YACCR for yacc -r
- YACCE for yacc -e
- LEX for lex.

The **make** program uses the following macro names to represent flags that it uses:

- CFLAGS for C compiler flags
- RFLAGS for ratfor compiler flags
- EFLAGS for efl compiler flags
- YFLAGS for yacc flags
- LFLAGS for lex flags.

Therefore, the command:

```
make "CC=newcc"
```

tells **make** to use the **newcc** program in place of the usual C language compiler. Similarly, the command:

```
make "CFLAGS=-O"
```

tells **make** to optimize the final object code produced by the C language compiler.

To look at the internal rules (in **rules.c**) that **make** uses, enter the following command:

```
make -p -f /dev/null 2>/dev/null
```

The output appears on the standard output.

Defining Default Conditions

When **make** creates a target file and cannot find commands in the description file and internal rules to create a file, it looks at the description file for default conditions. To define the commands that **make** performs in this case, use the .DEFAULT target name in the description file:

```
.DEFAULT:
        command
        command
          .
          .
          .
```

Because .DEFAULT is not a real target file, it is called a ***fake target***. Use the .DEFAULT fake target for an error recovery routine, or for a general procedure to create all files in the program that are not defined by an internal rule of the **make** program.

Including Other Files

Include files other than the current description file by using the word **include** as the first word on any line in the description file. Follow the word with a blank or a tab, and then the set of file names for **make** to include in the operation. For example:

```
include    /u/tom/temp /u/tom/sample
```

tells **make** to read the files temp, sample and the current description file to build the target file.

Do not use more than 16 levels of nesting with the include files feature.

Defining Macros

A macro is a name (or label) to use in place of several other names. It is a shorthand way of using the longer string of characters. To define a macro:

1. Start a new line with the name of the macro.
2. Follow the name with an = (equal sign).
3. To the right of the =, enter the string of characters that the macro name represents.

The macro definition can contain blanks before and after the = without affecting the result. The macro definition cannot contain a : (colon) or a tab before the =.

The following are examples of macro definitions:

```
# Macro "2" has a value of "xyz"
2 = xyz

# Macro "abc" has a value of "-ll -ly"
abc = -ll -ly

# Macro "LIBES" has a null value
LIBES =
```

A macro that is named but is not defined has a value of the null string.

Using Macros in a Description File

After defining a macro in a description file, use the macro in the description file commands by putting a $ (dollar sign) before the name of the macro. If the macro name is longer than one character, put () (parentheses) or { } (braces) around it. The following are examples of using macros:

```
$(CFLAGS)
$2
$(xy)
$Z
$(Z)
```

The last two examples have the same effect.

The following fragment shows how to define and use some macros:

```
# OBJECTS is the 3 files x.o, y.o and
# z.o (previously compiled)
OBJECTS = x.o y.o z.o

# LIBES is the standard library
LIBES = -lc

# prog depends on x.o y.o and z.o
prog:  $(OBJECTS)
# Link and load the 3 files with
# the standard library to make prog
      cc $(OBJECTS) $(LIBES) -o prog
```

The **make** program that uses that description file loads the three object files with the **libc.a** library.

A macro definition entered on the command line replaces any macro definitions in the description file that define the same macro label. Therefore, the command:

```
make "LIBES= -ll"
```

loads the files with the Lex (`-ll`) library.

Note: When entering macros with blanks in them on the command line, put " (double quotes) around the macro. Without the double quotes, the shell interprets the blanks as parameter separators and not a part of the macro.

Internal Macros

The **make** program has built-in macro definitions for use in the description file. These macros help specify variables in the description file. The **make** program replaces the macros with one of the following values:

$@	The name of the current target file
$$@	The label name on the dependency line
$?	The names of the files that have changed more recently than the target
$<	The name of the out-of-date file that caused a target file to be created
$*	The name of the current parent file without the suffix
$%	The name of an archive library member.

Target File Name

If the $@ macro is in the command sequence in the description file, **make** replaces the symbol with the full name of the current target file before passing the command to the shell to be run. The **make** program replaces the symbol only when it runs commands from the description file to create the target file.

Label Name

If the $$@ macro is on the dependency line in a description file, **make** replaces this symbol with the label name that is on the left side of the colon in the dependency line. This name could be a target file name, the name of a new flag, or the name of another macro. For example, if the following is included in a dependency line:

```
cat:      $$@.c
```

The **make** program translates it to:

```
cat:      cat.c
```

when **make** evaluates the expression. Use this macro to build a group of files, each of which has only one source file. For example, to maintain a directory of system commands, use a description file like:

```
# Define macro CMDS as a series
# of command names
CMDS = cat dd echo date cc cmp comm ar ld chown

# Each command depends on a .c file
$(CMDS):      $$@.c

# Create the new command set by compiling the out of
# date files ($?) to the target file name ($@)
        $(CC) -O $? -o $@
```

The **make** program changes the $$(@F) macro to the file part of $@ when it runs. For example, use this symbol when maintaining the **usr/include** directory while using a description file in another directory. That description file would look like:

```
# Define directory name macro INCDIR
INCDIR = /usr/include

# Define a group of files in the directory
# with the macro name INCLUDES
INCLUDES = \
        $(INCDIR)/stdio.h \
        $(INCDIR)/pwd.h \
        $(INCDIR)/dir.h \
        $(INCDIR)/a.out.h \

# Each file in the list depends on a file
# of the same name in the current directory
$(INCLUDES):       $$(@F)

# Copy the younger files from the current
# directory to /usr/include
        cp $? $@

# Set the target files to read only status
        chmod 0444 $@
```

This description file creates a file in the **/usr/include** directory when the corresponding file in the current directory has been changed.

Younger Files

Note: Before using the $? macro on a system with Distributed Services installed, ensure that the system clocks show the same date and time for all nodes that contain files for **make** to process.

If the $? macro is in the command sequence in the description file, **make** replaces the symbol with a list of parent files that have been changed since the target file was last changed. The **make** program replaces the symbol only when it runs commands from the description file to create the target file.

First Out-of-date File

Note: Before using the $< macro on a system with Distributed Services installed, ensure that the system clocks show the same date and time for all nodes that contain files for **make** to process.

If the $< macro is in the command sequence in the description file, **make** replaces the symbol with the name of the file that started the file creation. The file name is the name of the parent file that was out-of-date with the target file, and therefore caused **make** to create the target file again.

In addition, use a letter (D or F) after the < (less-than sign) to get either the directory name (D) or the file name (F) of the first out-of-date file. For example, if the first out-of-date file were:

```
/u/tom/sample.c
```

then **make** gives the following values:

```
$(<D)    =    /u/tom
$(<F)    =    sample
$<       =    /u/tom/sample
```

The **make** program replaces this symbol only when it runs commands from its internal rules or from the .DEFAULT list.

Current File Name Prefix

If the $* macro is in the command sequence in the description file, **make** replaces the symbol with the file name part (without the suffix) of the parent file that **make** is currently using to generate the target file. For example, if **make** is using the file:

```
test.c
```

then the $* represents the file name, `test`.

In addition, use a letter (D or F) after the * (asterisk) to get either the directory name (D) or the file name (F) of the current file.

For example, **make** uses many files (specified either in the description file or the internal rules) to create a target file. Only one of those files (the ***current file***) is used at any moment. If that current file were:

```
/u/tom/sample.c
```

then **make** gives the following values for the macros:

```
$(*D)   =  /u/tom
$(*F)   =  sample
$*      =  /u/tom/sample
```

The **make** program replaces this symbol only when it runs commands from its internal rules (or from the **.DEFAULT** list), and not when running commands from a description file.

Archive Library Member

If the $% macro is in a description file, and the target file is an archive library member, **make** replaces the macro symbol with the name of the library member. For example, if the target file is:

```
lib(file.o)
```

then **make** replaces the $% with the member name, `file.o`.

Changing Macro Definitions in a Command

When macros in the shell commands are in the description file, you can change the values that **make** assigns to the macro. To change the assignment of the macro, put a : (colon) after the macro name, followed by a replacement string. The form is as follows:

```
$(macro:string1=string2)
```

When **make** reads the macro and begins to assign the values to the macro from the macro definition, it replaces each `string1` in the macro definition with a value of `string2`. For example, if the description file contains the macro definition:

```
FILES=test.o sample.o form.o defs
```

you can replace the file `form.o` with a new file, `input.o`, by using the macro in the description file commands:

```
cc -o $(FILES:form.o=input.o)
```

Changing the value of a macro in this manner is useful when maintaining archive libraries (see the **ar** program in *AIX Operating System Commands Reference*).

Using make with SCCS Files

The **make** program does not allow references to prefixes of file names. Because SCCS file names begin with an **s.**, do not refer to them directly within a **make** description file. The **make** program uses a different suffix, the ~ (tilde), to represent SCCS files. Therefore, `.c~.o` refers to the rule that transforms an SCCS C language source file into an object. The internal rule is:

```
.c~.o:
        $(GET) $(GFLAGS) -p  $<  >$*.c
        $(CC) $(CFLAGS) -c $*.c
        -rm -f $*.c
```

The ~ added to any suffix changes the file search into an SCCS file name search with the actual suffix named by the . (period) and all characters up to (but not including) the ~. The `GFLAGS` macro passes flags to SCCS to determine the version of the SCCS files to be used.

The **make** program recognizes the following SCCS suffixes:

.c~	c source
.y~	yacc source grammar
.s~	assembler source
.sh~	shell
.h~	header

The **make** program has internal rules for changing the following SCCS files:

.c~:
.sh~:
.c~.o:
.s~.o:
.y~.o:
.l~.o:
.y~.c:
.c~.a:
.s~.a:
.h~.h:

Description Files Stored in SCCS

If you specify a description file, or a file named **makefile** is in the current directory, **make** does not look for a description file within SCCS. If a description file is not in the current directory and you enter the command, `make`, the **make** program looks for an SCCS file named either **s.makefile** or **s.Makefile**. If either of these files are present, **make** uses a **get** command to tell SCCS to build the description file from that source file. The value of the internal macro, **GETFLAGS**, determines the level of the file that SCCS creates. When SCCS creates the description file, **make** uses the file as a normal description file. When **make** finishes, it removes the created description file from the current directory.

How make Uses the Environment Variables

Each time **make** runs, it reads the current environment variables and adds them to its defined macros. In addition, it creates a new macro called **MAKEFLAGS**. This macro is a collection of all input flags to the **make** program (without the minus signs). Command line flags and assignments in the description file can also change the **MAKEFLAGS** macro. When **make** starts another process, it passes **MAKEFLAGS** to that process by using the **export** command.

When **make** runs, it assigns macro definitions in the following order:

1. Reads the **MAKEFLAGS** environment variable to set debug on, if it is needed.

 If **MAKEFLAGS** is not present or null, **make** sets its internal **MAKEFLAGS** variable to the null string. Otherwise, **make** assumes that each letter in **MAKEFLAGS** is an input flag. The **make** program uses these flags (except for the **-f, -p,** and **-r** flags) to determine its operating conditions.

2. Reads and sets the input flags from the command line. The command line adds to the previous settings from the **MAKEFLAGS** environment variable.

3. Reads macro definitions from the command line. **Make** ignores any further assignments to these names.

4. Reads the internal macro definitions.

5. Reads the environment, including the **MAKEFLAGS** macro. The **make** program treats the environment variables as macro definitions and passes them to other shell programs.

Example of a Description File

Figure 2-4 shows the description file that maintains the **make** program. The source code for **make** is spread over a number of C language source files and a Yacc grammar.

```
# Description file for the Make program

# Macro def: send to be printed
P = und -3 | opr -r2

# Macro def: source filenames used
FILES = Makefile version.c defs main.c
        doname.c misc.c files.c
        dosy.c gram.y lex.c gcos.c

# Macro def: object filenames used
OBJECTS = version.o main.o doname.o
          misc.o files.o dosys.o
          gram.o

# Macro def: lint program and flags
LINT = lint -p

# Macro def: C compiler flags
CFLAGS = -O

# make depends on the files specified
# in the OBJECTS macro definition
make:    $(OBJECTS)
# Build make with the cc program
        cc $(CFLAGS) $(OBJECTS) -o make
# Show the file sizes
        size make
```

Figure 2-4 (Part 1 of 3). Example Description File

```
# The object files depend on a file
# named defs
$(OBJECTS):  defs

# The file gram.o depends on lex.c
# uses internal rules to build gram.o
gram.o:  lex.c

# Clean up the intermediate files
clean:
        -rm *.o gram.c
        -du

# Copy the newly created program
# to /usr/bin and deletes the program
# from the current directory
install:
        @size make /usr/bin/make
        cp make /usr/bin/make ; rm make

# Empty file "print" depends on the
# files included in the macro FILES
print:   $(FILES)
# Print the recently changed files
        pr $? | $P
# Change the date on the empty file,
# print, to show the date of the last
# printing
        touch print
```

Figure 2-4 (Part 2 of 3). Example Description File

```
# Check the date of the old
# file against the date
# of the newly created file
test:
        make -dp | grep -v TIME >1zap
        /usr/bin/make -dp | grep -v TIME >2zap
        diff 1zap 2zap
        rm 1zap 2zap

# The program, lint, depends on the
# files that are listed
lint:   dosys.c doname.c files.c main.c misc.c
        version.c gram.c
# Run lint on the files listed
# LINT is an internal macro
        $(LINT) dosys. doname.c files.c main.c
        misc.c version.c gram.c
        rm gram.c

# Archive the files that build make
arch:
        ar uv /sys/source/s2/make.a $(FILES)
```

Figure 2-4 (Part 3 of 3). Example Description File

The **make** program usually writes out each command before issuing it.

The following output results from typing the simple command **make** in a directory containing only the source and description file:

```
cc -0 -c version.c
cc -0 -c main.c
cc -0 -c doname.c
cc -0 -c misc.c
cc -0 -c files.c
cc -0 -c dosys.c
yacc  gram.y
mv y.tab.c gram.c
cc -0 -c gram.c
cc version.o main.o doname.o misc.o files.o dosys.o
   gram.o -o make
13188+3348+3044 = 19580b = 046174b
```

Although none of the source files or grammars are specified in the description file, **make** uses its suffix rules to find them and issues the needed commands. The string of digits is the result of the **size make** command. The @ (at sign) on the **size** command in the description file prevented writing of the command, so only the sizes are written.

The output can be sent to a different printer or to a file by changing the definition of the **P** macro on the command line as follows:

```
make print "P = print -sp"
          or
make print "P = cat >zap"
```

Chapter 3. Using the Subroutine Libraries

CONTENTS

About This Chapter 3-2
System Libraries 3-3
 Including Declarations 3-7
 Linking the Library Routines 3-7
 Library Descriptions 3-7
The C Library 3-8
 Input/Output Control 3-8
 String Manipulation 3-15
 Memory Manipulation 3-18
 Character Manipulation 3-18
 Time 3-23
 Numerical Conversion 3-24
 Group File Access 3-25
 Password File Access 3-25
 Parameter Access 3-26
 Hash Table Management 3-26
 Binary Tree Management 3-26
 Table Management 3-27
 Memory Allocation 3-27
 Pseudo-random Number Generation 3-27
 Signal Handling 3-28
 Miscellaneous 3-29
Run-Time Services Library 3-30
Math Library 3-32
 Trigonometry 3-32
 Bessel 3-33
 Hyperbolic 3-33
 Miscellaneous 3-33
Network Management Library 3-34

About This Chapter

This chapter describes the subroutine libraries that are included with the AIX Operating System (AIX) and Japanese Language Support. Refer to *AIX Operating System Technical Reference* for complete technical information about each library function. This chapter groups the functions by library, and by function within the library. It describes the following commonly used libraries:

- C library (with or without hardware floating-point)
- Run-time services library
- Math library (with or without hardware floating-point)
- Ring Error Monitor Library.

Japanese Language Support Information

Note: The discussion of Japanese Language Support in *Japanese Language Support User's Guide.*

System Libraries

Figure 3-1 on page 3-4 lists the system libraries. These libraries are collections of commonly used functions and declarations. Use them in a program to avoid creating the functions for each new program.

To use the library functions:

- Include any declarations for the variables that the library routines use in the program
- Link the library routines with the program files after the program is compiled, or in the same process using the **cc** command.

Name	Path name	cc flag	Function
General C Libraries:			
C library	/lib/libc.a	Not required	Common C language subroutines for file access, string operations, character operations, memory allocation and other functions.
C library (floating-point)	/lib/libfc.a	-lfc	The same routines as the C library, except that some were compiled to use the hardware Floating-Point Accelerator to perform floating-point arithmetic.
Math library	/lib/libm.a	-lm	Mathematical functions using software routines to perform floating-point arithmetic.
Floating-Point Math library	/lib/libfm.a	-lfm	The same routines as the math library, except that they were compiled to use the hardware Floating-Point Accelerator to perform floating-point arithmetic.
Run-Time Services library	/usr/lib/librts.a	Not required	Support system services such as system configuration, messages, trace and error log support.
Programmer Workbench library	/lib/libPW.a	-lPW	Miscellaneous operating system functions.
Stand-alone Subroutine library	/usr/lib/lib2.a	-l2	Miscellaneous operating system functions.
Network Management library	/usr/lib/libipc.a	-lipc	Interprocess communication functions for Distributed Services

Figure 3-1 (Part 1 of 3). Summary of System Libraries

Name	Path name	cc flag	Function
Terminal I/O Libraries:			
Curses	/usr/lib/libcurses.a	-lcurses	Control functions for writing data to and getting data from the terminal screen.
extended curses	/usr/lib/libcur.a	-lcur	Control functions for writing data to and getting data from the terminal screen that support color, multiple windows, and an enhanced character set.
DOS Libraries:			
DOS library	/usr/lib/libdos.a	-ldos	DOS (Disk Operating System) functions for a program running on an RT work station.
DOS I/O library	/usr/lib/libdossio.a	-ldossio	Access to DOS (Disk Operating System) file systems and I/O functions for a program running on an RT work station.
Graphics Libraries:			
DASI 300 Graphics Interface library	/usr/lib/lib300.a	-l300	Graphics functions to a DASI 300 terminal.
DASI 300s Graphics Interface library	/usr/lib/lib300s.a, or /usr/lib/lib300S.a	-l300s, or -l300S	Graphics functions to a DASI 300s terminal.
DASI 450 Graphics Interface library	/usr/lib/lib450.a	-l450	Graphics functions to a DASI 450 terminal.
Tektronix 4014 Graphics Interface library	/usr/lib/lib4014.a	-l4014	Graphics functions to a Tektronix 4014 terminal.

Figure 3-1 (Part 2 of 3). Summary of System Libraries

Name	Path name	cc flag	Function
tplot Filters Graphics Interface library	/usr/lib/libplot.a	-lplot	Graphics functions for tplot filters.
IBM PC Graphics Printer Interface library	/usr/lib/libprint.a	-lprint	Graphics functions to an IBM 5250 Graphics Printer.
Other Libraries:			
Data Base Subroutine library	/usr/lib/libdbm.a	-ldbm	Database subroutines.
Queue Backend Subroutine library	/usr/lib/libqb.a	-lqb	Subroutines for queue backends.
Ring Error Monitor Library	/usr/lib/librem.a	-lrem	Subroutines to access the Ring Error Monitor.
sdb library	/usr/lib/libg.a	-lg	Subroutines that create object code with extra space for debugging with the **sdb** program.
lex library	/usr/lib/libl.a	-ll	Subroutines for programs created by the **lex** program generator.
yacc library	/usr/lib/liby.a	-ly	Subroutines for programs created by the **yacc** program generator.

Figure 3-1 (Part 3 of 3). Summary of System Libraries

Including Declarations

Some functions require a set of ***declarations*** to operate properly. You must specifically request that these declarations be included in a program. The system stores some declaration files, called ***header files***, in the **/usr/include** directory. To include a header file, use the following directive within a C language program:

```
#include <file.h>
```

where *file* is the name of one of the header files. Put all header file directives at the beginning of all files being compiled that use the header file.

Linking the Library Routines

When you compile a program, the **cc** program uses the **ld** program to search the C language library and the run-time services library to locate and include functions that are used in the program. To locate and include functions from other libraries, specify these libraries on the command line when starting the **cc** command. For example, when using functions of the math library, request that the math library be searched by including the argument

```
-lm
```

on the command line, like this :

```
cc file.c -lm
```

Use this method for all functions that are not part of the C language or run-time services libraries. Using this method, the compiler searches the C library and the run-time services library after searching the specified libraries. Refer to the description of the **ld** command in *AIX Operating System Commands Reference* for information about linking other libraries to a program.

Library Descriptions

The rest of this chapter describes the functions and header files of the libraries. Each library description begins with how to include the functions and/or header files in a program. Then, each function is listed and briefly described. Following the listing are descriptions of the header files associated with these functions (if any).

The C Library

The C library routines perform the following types of services:

- Input/output control
- String manipulation
- Character manipulation
- Time functions
- Miscellaneous functions.

The compiler loads the functions of the C library automatically. However, you must include any required declarations in the program.

Input/Output Control

The input and output (I/O) functions provide buffered I/O for a program that is easier to use than using the **read** and **write** system calls (see Chapter 4, "Using System Calls" on page 4-1). Do not specify any special flag to the compiler to use the I/O control functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <stdio.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

Using I/O Routines

The system treats devices as if they were files for input and output. Any of the I/O system calls or library routines can send data to or from either a device or a file. You must also open and close a device just as you do with a file.

Some of the I/O library routines are actually macros defined in a header file, and some are object modules of functions. In many cases the library contains a function and a macro that do the same type of operation. Consider the following points when deciding which to use:

- You cannot set a breakpoint for a macro using the debug program.
- Functions may have side effects to avoid.
- Macros usually are faster than the functions because the preprocessor replaces the macros with actual lines of code in the program.
- Macros result in larger object code after being compiled.

Some of the I/O routines use **stdin** and **stdout** as their input and output channel. Most of the routines, however, allow you to specify a file for the source or destination of the data transfer. Some specify the file using a file pointer (which points to a structure containing the file name); others accept a file descriptor (a positive integer assigned to the file when it is opened).

Figure 3-2 on page 3-10 summarizes some important characteristics of the input and output routines. The column headings mean:

Operation The name of the I/O routine or system call

Macro/Function If the operation is a library routine, this column shows if it is a macro or a function. If it is a system call, this column shows that it is a system call.

Input/Output The source and/or destination of the operation is either a file that you can **Specify**, or it uses **stdio** (standard input and output).

Formatted The resulting data stream is formatted (**Yes**) or not formatted (**No**).

Operation Type The data type of the information being transferred: byte, character (1-byte **char**), extended character (2-byte **NLchar**), word (4-bytes) or string. Some conversion operations accept one type of information as input and then output a different type of information.

Operation	Macro/Function	Input/Output	Formatted	Operation Type
read	System Call	Specify	No	Byte
write	System Call	Specify	No	Byte
fread	Function	Specify	No	Byte
fwrite	Function	Specify	No	Byte
printf	Function	stdio	Yes	Byte
fprintf	Function	Specify	Yes	Byte
sprintf	Function	Specify	Yes	Byte
scanf	Function	stdio	Yes	Byte
fscanf	Function	Specify	Yes	Byte
NLprintf	Function	stdio	Yes	NLchar
NLfprintf	Function	Specify	Yes	NLchar
NLsprintf	Function	Specify	Yes	NLchar
NLscanf	Function	stdio	Yes	NLchar
NLfscanf	Function	Specify	Yes	NLchar
ungetc	Function	Specify	No	Character
getc	Macro	Specify	No	Character
getchar	Macro	stdio	No	Character
fgetc	Function	Specify	No	Character
putc	Macro	Specify	No	Character
putchar	Macro	stdio	No	Character
fputc	Function	Specify	No	Character
getw	Function	Specify	No	Word
putw	Function	Specify	No	Word
gets	Function	stdio	No	String
fgets	Function	Specify	No	String
puts	Function	stdio	No	String
fputs	Function	Specify	No	String
sscanf	Function	Specify	Yes	String
NLsscanf	Function	Specify	Yes	NLchar string

Figure 3-2. Comparison of I/O Operations

Operation	Macro/Function	Input/Output	Formatted	Operation Type
getwc	Function	Specify	No	NLchar
fgetwc	Function	Specify	No	NLchar
getwchar	Function	stdio	No	NLchar
ungetwc	Function	Specify	No	NLchar
putwc	Function	Specify	No	NLchar
fputwc	Function	Specify	No	NLchar
putwchar	Function	stdio	No	NLchar
getws	Function	stdio	No	NLchar string
fgetws	Function	Specify	No	NLchar string
putws	Function	stdio	No	NLchar string
fputws	Function	Specify	No	NLchar string

Figure 3-3. Comparison of Japanese Language Support I/O Operations

Note: Entries in Figure 3-3 are active only when Japanese Language Support is installed on your system.

I/O Routines Descriptions

The I/O routine descriptions are grouped into the following categories:

- File access
- File status
- Input
- Output
- Miscellaneous.

In the following descriptions, ***stream*** input and output refers to sequential input and output using open file descriptors. The terms ***stdin*** and ***stdout*** refer to the device or file that is currently assigned as standard input or standard output.

File Access

fclose Closes an open stream.

fdopen Associates stream with an opened file.

fileno Returns a file descriptor associated with an open stream.

fopen Opens a stream with specified permissions. A stream is what **fopen** returns.

freopen Substitutes named file in place of open stream.

fseek	Repositions stream pointer.
pclose	Closes a stream opened by **popen**.
popen	Creates a pipe as a stream between two processes.
rewind	Repositions stream pointer at beginning of file.
setbuf	Turns buffering to stream on and off.

File Status

clearerr	Resets error condition on stream.
feof	Tests for end of file on stream.
ferror	Tests for error condition on stream.
ftell	Returns current stream pointer.

Input

fgetc	Reads next character from stream (function for the macro **getc**).
fgets	Reads string from stream.
fread	Reads from stream, buffered.
fscanf	Reads using format from stream.
getc	Returns next character from stream.
getchar	Returns next character from **stdin**.
gets	Reads string from **stdin**.
getw	Reads word from stream.
scanf	Reads using format from **stdin**.
sscanf	Reads using format from string.
NLscanf	Reads using format from **stdin**.
NLsscanf	Reads using format from string.
ungetc	Puts back one character on stream.

	Japanese Language Support Information
fgetwc	Reads next wide character from stream.
fgetws	Reads wide-character string from stream.
getwc	Returns next wide character from stream.
getwchar	Returns next wide character from **stdin**.
getws	Reads wide-character string from **stdin**.
ungetwc	Puts back one wide character on stream.

Output

fflush	Writes all currently buffered characters from stream.
fprintf	Writes using format to stream.
fputc	Writes next character to stream (function for **putc**).
fputs	Writes string to stream.
fwrite	Writes to stream, buffered.
NLfprintf	Writes using format to stream.
NLprintf	Writes using format to **stdout**.
NLsprintf	Writes using format to string.
printf	Writes using format to **stdout**.
putc	Writes next character to stream.
putchar	Writes next character to **stdout**.
puts	Writes string to **stdout**.
putw	Writes word to stream.
sprintf	Writes using format to string.

Japanese Language Support Information	
fputwc	Writes next wide character to stream.
fputws	Writes wide-character string to stream.
putwc	Writes next wide character to stream.
putwchar	Writes next wide character to **stdout**.
putws	Writes wide-character string to **stdout**.

Miscellaneous

ctermid	Returns file name for controlling terminal.
cuserid	Returns login name for owner of current process.
mktemp	Creates file using template and Distributed Services considerations.
NLgetfile	Gets parameter file for international character support.
NLgetctab	Gets collation table for international character support.
system	Executes system command.
tempnam	Creates temporary file name using node identifiers.
tmpnam	Creates temporary file name using node identifiers.
tmpfile	Creates temporary file using node considerations.

I/O Header File

The I/O header file is **stdio.h** in the **/usr/include** directory. This file contains macro definitions and parameters that the I/O library routines use. The shell automatically opens the following files:

stdin	Standard input file
stdout	Standard output file
stderr	Standard error file.

String Manipulation

The string manipulation functions include:

- Locate a character position within a string
- Locate a sequence of characters within a string
- Copy a string
- Concatenate strings
- Compare strings
- Translate a string
- Measure a string.

You do not need to specify any special flag to the compiler or include a header file for these functions in the program to use the string functions.

The **regexp** functions, **compile**, **step** and **advance**, require that the **regexp.h** header file be included. The **regcmp** and **regex** functions are in **libPW.a**. Other string functions are in **libc.a**, and require the inclusion of the **string.h** header file. The string routines perform the following functions:

compile	Compiles a regular expression using the current collating sequence.
step	Using a variable set by **compile**, matches a regular expression to the beginning of a string. Supports compiling a number of regular expressions (using **compile**) before attempting a match.
advance	Advances recursively to match the rest of the string to the rest of the expression, supporting metacharacters in a regular expression. The **advance** function supports the **compile** and **step** functions.
regcmp	Compiles a regular expression using the current collating sequence for international character support.
regex	Executes a compiled regular expression against a string using an ASCII collating sequence.
NCdecstr	Converts a string of **chars** to **NLchars**.
NCencstr	Converts a string of **NLchars** to **chars**.
strcat	Concatenates two strings.
NCstrcat	Concatenates two strings of **NLchars**.
NLstrcat	Concatenates two strings of **chars**.
strchr	Searches string for character.

NCstrchr	Searches string for **NLchar**.
NLstrchr	Searches string for **char**.
strcmp	Compares two strings using internal representations.
NCstrcmp	Compares two strings of **NLchar**s using the current collation sequence.
NLstrcmp	Compares two strings of **char**s using the current collation sequence.
strcpy	Copies string over string.
NCstrcpy	Copies **NLchar** string over **NLchar** string.
NLstrcpy	Copies string over string.
strcpsn	Returns the length of initial string not containing the compared set of characters.
NCstrcpsn	Returns the length in bytes of initial string not containing the compared set of **NLchar**s.
NLstrcpsn	Returns the length in bytes of initial string not containing the compared set of code points.
strlen	Returns the length of string.
NCstrlen	Returns the length of string of **NLchar**s.
NLstrlen	Returns the length of string in bytes.
NLstrdlen	Counts the length of string in code points.

Japanese Language Support Information	
NLstrdlen	Returns the display length of string in bytes.
NLcplen	Counts the length of string in code points.

strncat	Concatenates up to a specified number of bytes from one string to another string.
NCstrncat	Concatenates up to a specified number of **NLchar**s from one string to another string.
NLstrncat	Concatenates up to a specified number of bytes from one string to another string.
strncmp	Compares up to a specified number of bytes from one string with another string.
NCstrncmp	Compares up to a specified number of **NLchar**s from one string with another string.

NLstrncmp	Compares up to a specified number of bytes from one string with another string.
strncpy	Copies up to a specified number of bytes from one string to another string.
NCstrncpy	Copies up to a specified number of **NLchar**s from one string to another string.
NLstrncpy	Copies up to a specified number of bytes from one string to another string.
strpbrk	Searches string for any of a set of bytes.
NCstrpbrk	Searches code point string for the first occurrence of any of a set of **char**s.
NLstrpbrk	Searches string for the first occurrence of any of a set of code points.
strrchr	Searches string backwards for character.
NCstrrchr	Searches string for the last occurrence of an **NLchar**.
NLstrrchr	Searches string for the last occurrence of a code point.
strspn	Returns the length of an initial string containing a set of bytes.
strstr	Locates a sequence of characters (excluding the terminating **NULL** character) in a string
NCstrspn	Returns the length of an initial string of **NLchar**s containing the set of code points.
NLstrspn	Returns the length in bytes of an initial string containing the set of code points.
strtok	Searches string for a token separated by any of a set of bytes.
NCstrtok	Searches string for a token separated by any of a set of code points.
NLstrtok	Searches string for a token separated by any of a set of code points.
NLescstr	Converts a string possibly containing extended characters into ASCII bytes, preserving character information by converting each **NLchar** into a mnemonic string of ASCII bytes.
NLunescstr	Converts a string of ASCII bytes, possibly containing escape sequences representing extended characters, into a string in which any escape sequences are converted to **NLchar**s.
NLflatstr	Converts a character string possibly containing extended characters into ASCII bytes, preserving character appearance by converting each code point to an ASCII character it resembles. **Note: NLflatstr** is not active when Japanese Language Support is installed.

Memory Manipulation

The memory functions operate on arrays of characters in memory called ***memory areas***. The memory manipulation functions include:

- Locating a character within a memory area
- Copying characters between memory areas
- Comparing contents of memory areas
- Setting a memory area to a value.

You do not need to specify any special flag to the compiler to use the memory functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <memory.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

The memory routines perform the following functions:

memccpy Copies characters from one memory area to another, stopping at the first occurrence of a specified character or after a specified number of characters are copied.

memcpy Copies a specified number of characters from one memory area to another.

memchr Finds the first occurrence of a specified character in a memory area, and returns a pointer to that character.

memcmp Compares the contents of two memory areas up to a specified maximum number of characters.

memset Sets the contents of a memory area to a specified value.

Character Manipulation

The character manipulation functions and macros test and translate ASCII characters.

The character manipulation functions and macros are grouped into the following categories:

- Character testing
- Character translation
- Character collation.

Character Testing

Use the following functions and macros to determine character type. Punctuation, alphabetic, and case querying functions for **NLchar** values may vary depending on the current collation table.

isalnum	Is character alphanumeric?
isalpha	Is character alphabetic?
isascii	Is character an integer ASCII character?
iscntrl	Is character a control character?
isdigit	Is character a digit?
isgraph	Is character a printing character, not including space?
islower	Is character a lowercase letter?
isprint	Is character a printing character, including space?
ispunct	Is character a punctuation character?
isspace	Is character a white space character?
isupper	Is character an uppercase letter?
isxdigit	Is character a hex digit?
NCchrlen	What is the length of the code point in bytes?
NCdechr	(Macro) What is the **NLchar**?
NCeqvmap	Is the character an **NLchar** that begins an ***equivalence class***? An equivalence class is a class of **NLchars** that can be treated as identical in some collating contexts. Returns 1 if it is, and 0 if it is not. **NLeqvmap** is not available when Japanese Language Support is installed.
NCisalnum	Is **NLchar** an alphabetic character or digit?
NCisalpha	Is **NLchar** an alphabetic character?
NCiscntrl	Is **char** or **NLchar** an ASCII delete character (0177) or an ordinary ASCII control character other than the single-shift characters that indicate an extended character? (Extended character indicators are not available when Japanese Language Support is installed.)

NCisdigit	Is **NLchar** a decimal digit?
NCisgraph	Is **NLchar** a printing character, excluding the space character?
NCislower	Is **NLchar** a lowercase alphabetic character?
NCisNLchar	Is character a valid **NLchar**?
NCisprint	Is **NLchar** a printing character, including the space character?
NCispunct	Is **NLchar** a punctuation character?
NCisshift	Is **char** or **NLchar** the first byte of an extended character?
NCisspace	Is **NLchar** a space, tab, carriage return, new-line, vertical tab, or form-feed character?
NCisupper	Is **NLchar** an uppercase alphabetic character?
NCisxdigit	Is **NLchar** a hexadecimal digit?
NLchrlen	(Macro) How many bytes is the (possibly extended) character?
NLisNLcp	(Macro) If character is an extended character, how many bytes? (Returns the length of the code point, or zero if the code point is invalid.)

	Japanese Language Support Information
	The following functions and macros are also available.
isjalpha	Is character an alphabetic SJIS character?
isjdigit	Is character an Arabic SJIS digit?
isjgraph	Is character a printing SJIS character, not including space?
isjhira	Is character a hiragana character?
isjkata	Is character a katakana character?
isjkanji	Is character a kanji character?
isjlower	Is character a lowercase SJIS character?
isjparen	Is character a bracketing SJIS character?
isjprint	Is character a printing SJIS character, including space?
isjpunct	Is character a SJIS punctuation character?
isjspace	Is character a white space SJIS character?
isjupper	Is character an uppercase SJIS letter?
isjxdigit	Is character an Arabic hex SJIS digit?

Character Translation

Use these functions to translate characters from one form to another:

toascii	Converts integer to ASCII character.
tolower	Converts character to lowercase.
toupper	Converts character to uppercase.
NCdecode	Converts **char** to **NLchar**.
NCencode	Converts **NLchar** to **char**.
NCdec	(Macro) converts **char** to **NLchar**.
NCenc	(Macro) converts **NLchar** to **char**.

Japanese Language Support Information	
	These functions are also available:
cjistosj	Converts JIS character to SJIS character.
cjistouj	Converts JIS character to Unix-JIS character.
csjtouj	Converts SJIS character to Unix-JIS character.
csjtojis	Converts SJIS character to JIS character.
cujtojis	Converts UNIX-JIS character to JIS character.
cujtosj	Converts UNIX-JIS character to SJIS character.
jistosj	Converts JIS string to SJIS string.
jistouj	Converts JIS string to UNIX-JIS string.
sjtouj	Converts SJIS string to UNIX-JIS string.
sjtojis	Converts SJIS string to JIS string.
ujtojis	Converts UNIX-JIS string to JIS string.
ujtosj	Converts UNIX-JIS string to SJIS string.
tojhira	Converts SJIS katakana character to its SJIS equivalent hiragana character.
tojkata	Converts SJIS hiragana character to its SJIS equivalent katakana character.
atosjis	Converts ASCII character to SJIS character.
sjistoa	Converts SJIS character to ASCII character.
tojlower	Converts an uppercase kanji ASCII-equivalent character to a lowercase kanji ASCII-equivalent character.
tojupper	Converts a lowercase kanji ASCII-equivalent character to an uppercase kanji ASCII-equivalent character.

Character Collation

Redefinable character collation is a feature provided for international character support. Collation is performed by macros using a collation file created from a collation table by **ctab** command.

NCcollate	Returns the new collating value of the character for which it is called.
NCcoluniq	Assigns a unique sequential value to characters in equivalence classes, so that all characters have a unique value for some purpose.

NCeqvmap	Returns a nonzero value if the corresponding **NLchar** begins an equivalence class. Otherwise 0 is returned. **NCeqvmap** is not available when Japanese Language Support is installed.

Japanese Language Support Information

NCcolval	Returns the collation value for a character; this value is not necessarily unique.

Character Header File

The character header file is **ctype.h** in the **/usr/include** directory. It contains the macro definitions and data declarations that the ASCII string functions use.

Time

The time functions access and reformat the current system date and time. You do not need to specify any special flag to the compiler to use the time functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <time.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

These functions (except **tzset**) convert a time such as the time returned by the **time** system call:

asctime	Returns string representation of date and time.
ctime	Returns string representation of date and time, given integer form.
gmtime	Returns Greenwich Mean Time.
localtime	Returns local time.
tzset	Sets time zone field from environment variable.

Time Header File

The header file for the time functions is **time.h** in the **/usr/include** directory. It includes declarations for variables that the time functions use, such as:

tm A structure that the **gmtime** and **localtime** functions return.

daylight An integer that is nonzero to use Daylight Savings Time conversions.

tzname A character that defines the name of time zones. The system overrides this variable if the **TZ** variables are defined in the system environment. Setting the **TZ** variable changes the values defined in the header file for **daylight**, **timezone** and **tzname**.

Functions that define a large number of time-related variables are also supported in the **time.h** header file. For example:

ctime Converts the structure for time (a value in seconds since 00:00:00 Greenwich Mean Time, January 1, 1970) into a character string for day, date, and time.

NLstrtime Using the structures of **ctime**, formats time and date data into strings using an international character support format established by environment variables.

NLtmtime Takes a string and sets a time structure. The string data is handled in formats established by environment variables for international character support.

Numerical Conversion

These functions perform numerical conversion. You do not need to specify any special flag to the compiler or include a header file to use these functions.

a64l Converts string to base 64 ASCII.

atof Converts string to floating.

atoi Converts string to integer.

atol Converts string to long.

frexp Splits floating into mantissa and exponent.

l3tol Converts 3-byte integer to long.

ltol3 Converts long to 3-byte integer.

ldexp Combines mantissa and exponent.

l64a Converts base 64 ASCII to string.

modf Splits mantissa into integer and fraction.

Group File Access

These functions access the group file. You do not need to specify any special flag to the compiler to use these functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <grp.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

endgrent Closes group file being processed.

getgrent Gets next group file entry.

getgrgid Returns next group with matching gid.

getgrnam Returns next group with matching name.

setgrent Rewinds group file being processed.

Password File Access

These functions search and access information stored in the password file **/etc/passwd**.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <pwd.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

endpwent Closes password file being processed.

getpw Searches password file for user ID.

getpwent Gets next password file entry.

getpwnam Returns next entry with matching name.

getpwuid Returns next entry with matching user ID.

putpwent Writes entry on stream.

setpwent Rewinds password file being accessed.

Parameter Access

These functions get several different types of parameters from the system. You do not need to specify any special flag to the compiler or include any header file to use these functions.

getopt Gets next option from option list in command.

getcwd Returns string representation of current directory.

getenv Returns string value associated with environment variable.

NLgetenv Returns string value associated with international character support environment variable.

getpass Reads string from terminal without echoing.

Hash Table Management

These functions manage hash search tables. You do not need to specify any special flag to the compiler or include any header file to use these functions.

hcreate Creates hash table.

hdestroy Destroys hash table.

hsearch Searches hash table for entry.

Binary Tree Management

These functions manage a binary tree. You do not need to specify any special flag to the compiler or include any header file to use these functions.

tdelete Deletes nodes from binary tree.

tsearch Searches binary tree.

twalk Walks through a binary tree to a specified level, and performs a specified action at each node of the tree.

Table Management

These functions manage a table. The table is a two-dimensional character array. The first subscript defines the maximum number of entries in the table. The second subscript defines the width (or length) of a single entry. These functions do not allocate storage. Be sure to allocate sufficient memory before using these functions.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

bsearch Searches table using binary search.

lsearch Searches table using linear search.

qsort Sorts table using quicker-sort algorithm.

Memory Allocation

These functions allocate or free memory from the program.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

calloc Allocates zeroed storage.

free Frees previously allocated storage.

malloc Allocates storage.

realloc Changes size of allocated storage.

Pseudo-random Number Generation

These functions generate pseudo-random numbers. The functions that end with 48 use a pseudo-random number generator based upon the linear congruent algorithm and 48-bit integer arithmetic. The **rand** and **srand** functions use a multiplicative congruent random number generator with period of 2^{32}.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

drand48 Returns a random double, n, in the interval:

$$0 \leq n < 1$$

lcong48 Sets parameters for **drand48**, **lrand48**, and **mrand48**.

lrand48 Returns a random long, n, in the interval:

$0 \leq n < 2^{31}$

mrand48 Returns a random long, n, in the interval:

$-2^{31} \leq n < 2^{31}$

rand Returns a random integer, n, in the interval:

$0 \leq n < 2^{15}$

seed48 Seeds the generator for **drand48**, **lrand48**, and **mrand48**.

srand Seeds the generator for rand.

srand48 Seeds the generator for **drand48**, **lrand48**, and **mrand48**.

Signal Handling

These functions simulate the functions available from the signal handling functions provided by the system calls for signals described in "Signal Calls" on page 4-29. These functions indicate error handling to other processes, and communicate with other cooperating processes.

You do not need to specify any special flag to the compiler to use these functions.

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <signal.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

These declarations define ASCII names for each of the software signals.

gsignal Sends a software signal.

ssignal Arranges for handling of software signals.

Miscellaneous

These functions perform services that do not appear in any of the previous categories.

You do not need to specify any special flag to the compiler or include any header file to use these functions.

abort Sends an IOT (I/O terminate) signal to the process.

abs Returns the absolute integer value.

ecvt Converts double to string.

fcvt Converts double to string using FORTRAN format.

gcvt Converts double to string using FORTRAN F or E format.

isatty Tests whether integer file descriptor is associated with a terminal.

monitor Causes process to record a histogram of program counter location.

swab Swaps and copies bytes.

ttyname Returns the path name of terminal associated with integer file descriptor.

Run-Time Services Library

The run-time services library routines allow you to access the following system functions from your program:

- Configuration services
- Message services
- Trace
- Error logging.

The functions are in the file **librts.a** in the **/usr/lib** directory. The **cc** command automatically locates and loads the requested functions when it compiles a C language program.

Include header files, as needed, when using these routines. See *AIX Operating System Technical Reference* for the header files needed with each routine, as well as detailed information about their use.

cfgadev	Adds a device.
cfgamni	Adds a minidisk.
cfgaply	Applies configuration information to the kernel.
cfgcadz	Adds or replaces a stanza in an attribute file.
cfgcclsf	Closes an attribute file.
cfgcdlsz	Deletes a stanza from an attribute file.
cfgcopsf	Opens an attribute file.
cfgcrdsz	Reads an attribute file stanza.
cfgddev	Deletes a device.
cfgdmni	Deletes a minidisk.
errunix	Logs errors that occur when running a program.
mdverify	Controls write-verify operation for a minidisk.
msghelp	Retrieves and displays a predefined help message.
msgimed	Retrieves and outputs a predefined immediate message.
msgqued	Retrieves and outputs a predefined queued message.
msgrtrv	Retrieves text for a message, insert or help.
trace_on	Checks whether trace channel is enabled.

trc_start	Lets a process start a trace daemon.
trc_stop	Lets a process stop a trace daemon.
trcunix	Records trace log entries for a program.
vrcppr	Installs a protocol procedure.

The library also contains other routines that these routines use to perform their functions.

Math Library

The math library consists of functions and a header file. Use the following command line entry to tell the **cc** command to locate and load the needed functions when it links a C language program:

```
cc file.c -lm
```

Include the header file for these functions in the program. To include a header file, use the following statement:

```
#include <math.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

The math header file is **math.h** in the **/usr/include** directory. This file contains definitions for functions and macros that the math library routines use. All functions in the math library return double precision values.

The functions are grouped into the following categories:

- Trigonometric functions
- Bessel functions
- Hyperbolic functions
- Miscellaneous functions.

Trigonometry

These functions compute angles (in decimal radian measure), sines, cosines, and tangents. All of these values are expressed in double precision. The file **math.h** declares the values as **double**.

acos	Returns arc cosine.
asin	Returns arc sine.
atan	Returns arc tangent.
atan2	Returns arc tangent of a ratio.
cos	Returns cosine.
hypot	Returns the square root of the sum of the squares of two numbers.
sin	Returns sine.
tan	Returns tangent.

Bessel

These functions calculate bessel functions of the first and second kinds of several orders for real values. The bessel functions are:

j0, j1, jn, y0, y1, and yn

For descriptions of the functions see **bessel** in *AIX Operating System Technical Reference.*

Hyperbolic

These functions compute the hyperbolic sine, cosine, and tangent for real values.

cosh Returns hyperbolic cosine.

sinh Returns hyperbolic tangent.

tanh Returns hyperbolic tangent.

Miscellaneous

These functions do not fall into any of the previously defined categories.

ceil Returns the smallest integer not less than a given value.

exp Returns the exponential function of a given value.

fabs Returns the absolute value of a given value.

floor Returns the largest integer not greater than a given value.

fmod Returns the remainder produced by the division of two given values.

gamma Returns the natural log of gamma as a function of the absolute value of a given value.

log Returns the natural logarithm of a given value.

pow Returns the result of a given value raised to another given value.

sqrt Returns the square root of a given value.

Network Management Library

These functions provide information about network IDs, and allow you to map IPC keys to other remote or local IPC keys. The functions are in the file **libipc.a** in the **/usr/lib** directory. Programs that use these functions must be compiled using the `-lipc` flag for the **cc** command.

Include the header file for these functions in the program using the following statement:

```
#include <drs.h>
```

All include statements should be near the beginning of the first file being compiled, usually in the declarations section before `main( )`, and must occur before using any library functions.

create_ipc_prof Creates a profile for an IPC queue used with or without Distributed Services.

del_ipc_prof Deletes a profile for an IPC queue used with or without Distributed Services.

drsname Returns the node ID for a supplied node nickname.

drsnidd Returns the node nickname for a supplied node ID.

find_ipc_prof Finds a profile for an IPC queue from supplied information (name and/or key).

Chapter 4. Using System Calls

CONTENTS

About This Chapter 4-2
Header Files Needed for Calls 4-2
Process Calls 4-4
 Process Handling Calls 4-4
 Process Identification Calls 4-18
 Process Tracking Calls 4-29
Interprocess Communications 4-29
 Signal Calls 4-29
 Enhanced Signal Facility 4-34
 Semaphore Calls 4-42
 Message Calls 4-55
System Memory Management 4-67
 Segment Registers 4-69
 Using a Mapped Data File 4-70
 Using a Mapped Executable File 4-72
 Shared Memory Calls 4-74
 Memory Management Calls 4-77
File System Calls 4-78
 Data Handling Calls 4-78
 File Maintenance Calls 4-83
Time System Calls 4-84
Using System Calls with Distributed Services 4-85
 Distributed Services Example Program 4-86
 Using File Locking 4-93
 Distributed Services Message Queue Example 4-100

About This Chapter

This chapter discusses how to access the services of the operating system from a program through ***system calls***. See *AIX Operating System Technical Reference* for reference information about a system call described in this chapter.

Header Files Needed for Calls

Some system calls depend on special macro definitions and declarations for the values that they return. The system provides this information in files called ***header files*** that are in the directory **/usr/include**. Therefore, when using system calls, ensure to also include any header files that the system call needs. To include a file in a program, use the following statement in the program:

```
#include <file.h>
```

where the parameter, *file.h*, represents the name of the header file to use.

The header files needed for each system call are:

Header File	Calls That Use The Header File
fcntl.h	open, fcntl
grp.h	getgroups, setgroups
lockf.h	lockf
mon.h	profil
signal.h	sigsetmask, sigstack
sys/chownx.h	chownx
sys/devinfo.h	ioctl
sys/dsstate.h	dsstate
sys/fullstat.h	fullstat, ffullstat
sys/ioctl.h	ioctl
sys/ipc.h	msgxrcv, msgctl, msgget, msgsnd, msgrcv, semctl, semget, semop, shmat, shmctl, shmdt, shmget
sys/dstables.h	loadtbl

Header File	Calls That Use The Header File
sys/lock.h	plock
sys/msg.h	msgxrcv, msgctl, msgget, msgsnd, msgrcv
sys/mntctl.h	mntctl
sys/reg.h	ptrace
sys/select	select
sys/sem.h	semctl, semget, semop
sys/shm.h	disclaim, shmctl, shmget, shmat, shmdt
sys/signal.h	signal, sigvec
sys/stat.h	stat, fstat, creat, chmod, msgget, mknod, shmget, semget
sys/times.h	times
sys/types.h	lseek, msgxrcv, msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmat, shmctl, shmdt, shmget, times, ulimit, ustat
sys/utsname.h	uname
sys/vmount.h	mount, vmount
uinfo.h	usrinfo
unistd.h	access, utime
ustat.h	ustat

Process Calls

When a program runs in the system, that program, together with the environment that it runs in, is a process. Many processes are running in the system: some running system programs (like **init**) and some running application programs. When each process begins, the system assigns it an identification number (***process ID***) that is a positive integer between 0 and 32,767. As long as the process remains active, the system uses this number to identify that process. When the process ends, the system can assign the number to a new process.

Process Handling Calls

Use the following calls to control creating, operating and stopping processes (see *AIX Operating System Technical Reference* for more information about these calls):

Call	Description
exec	Runs a new program in the currently running process.
exit	Stops a process.
fork	Creates a new process.
nice	Changes priority of a process.
pipe	Creates an inter-process channel.
plock	Locks process, text, or data in memory.

Starting a Process

All processes that run in the system, except **init**, are started through the **fork** system call. The **fork** call creates a second independent process from the one running process through the following sequence of events:

1. Gets a new process ID from the system.
2. Creates a new process that has access to the code running in the existing process and the environment of the existing process. By using virtual memory mapping, the system does not actually copy the information into the new process until the new process actually uses it. If, for example, the first instruction executed in the new process is an **exec** system call, the system does not waste time copying the first program only to throw it out when the **exec** call occurs.
3. Starts the new process running from the place in the program that immediately follows the fork call.

The original process is called the ***parent***; the new process is called the ***child***. The value returned to a process from the **fork** call tells that process whether it is a child or a parent process. The parent process receives the process ID of the child process; the child process

receives a value of zero. Therefore, in most programs that use a **fork** system call, it appears within a conditional statement.

```
if( x = fork() )
{
        [code executed by the parent process]
}
else
{
        [executed by the child process]
        execv( newprog );
}
```

The parent process may continue running and create more new processes with other **fork** calls. However, to ensure that the child process completes successfully, the parent process usually stops running until the child process completes its assigned task. The parent process uses the **wait** system call to stop until the child process ends. When used in this manner, **fork** and **wait** are similar to calling a subroutine -- the calling program waits for the subroutine to complete and return a value before continuing.

If the fork operation is successful, the first instruction that the child process executes is usually an **exec** system call. The **exec** call replaces the program currently running in the child process with a new program and starts executing the new program.

When the child process completes, the parent process receives a completion code to indicate the conditions of the child process at the end. A value of zero indicates a normal end; a value that is not zero indicates an error (defined by the value of **errno**, the system variable that reports error codes).

Note: If the parent process does not wait for a completion code from the child process, and the child ends with an error condition, no other process gets the completion code to recover from the error. If the parent does not wait for a completion code when it creates the child process, it can continue to perform operations as a separate process. After taking care of some other tasks, the parent can then issue a **wait** call. If the child process has not ended before the **wait** occurs, the parent still gets the completion code when the child process ends.

Example of Process Life Cycle

When the shell executes a command, it performs the fork and wait process just described. See Figure 4-1 on page 4-7 for an illustration of the following process. For example, the command:

```
pr
```

tells the shell to execute a program called **pr**. To do this:

1. The shell issues a **fork** system call to get a new process.
2. The system marks the shell data and stack space as ***copy on reference*** for the new process space.
3. The system schedules the new process to be run.
4. The first shell issues a **wait** to wait for the new process (child) to complete.

The new process (also a shell) then:

1. Issues an **exec** system call to load and run the **pr** program in the new process
2. Issues an **exit** system call with a completion code when it is done.

When the parent shell process receives the completion code from the **exit** system call in the child, it starts running again. You can then enter another command on the command line.

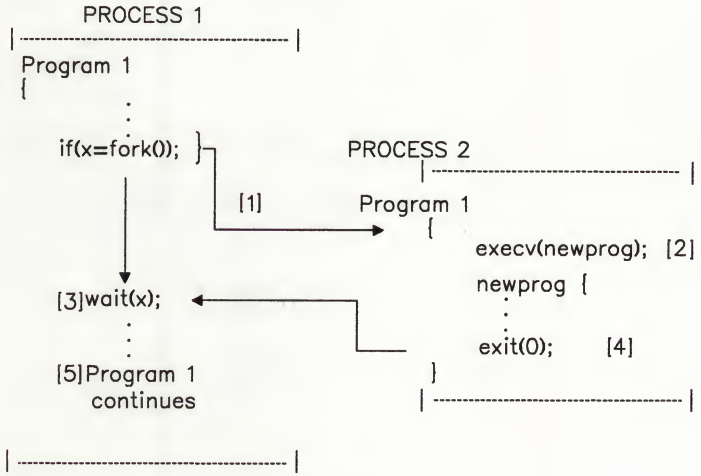


Figure 4-1. Using the Fork System Call

Special Processes

All processes in the system begin from a fork. One special process, process ID 1, is the initialization process (**init**). This process is the ancestor process to all other processes in the system. It uses the **exec** system call to start the remaining system programs.

Example of Fork System Call

The program in Figure 4-2 on page 4-8 uses the **fork** system call in a simple, but operating program. If you are using the Programming Examples, the program is stored as `forktst1.c`. Compile the program using the **cc** command:

```
cc forktst1.c -o forktst1
```

To run the program, enter the command:

forktst1

When the program runs, it writes output to the screen. The two processes that the program creates (parent and child) both write to the screen without regard to the other process. Therefore, the result that appears on the screen depends upon system activity and random luck. Sometimes it is a perfect output like in Figure 4-3 on page 4-9; other times the output from the two processes mix together on the screen. Run the program several times to experiment with the operation of two concurrent processes.

```
#include <stdio.h>
#define PR(value)  fprintf(stdout,"%s\n",(value) )

main()
{
  int c_count;
  int p_count;

  PR("starting main process");
  if (fork() == 0)
  {
     PR("starting child process");
     for(c_count = 0; c_count <= 4; c_count++)
        PR("child");
     PR("ending child process");
     exit(0);
  }
  PR("starting parent process");
  for(p_count = 0; p_count <= 4; p_count++)
     PR("parent");

  PR("ending parent process");
}
```

Figure 4-2. Example of Fork System Call

Well Behaved Output	Possible Output
starting main process	starting main process
starting parent process	starting parent process
parent	parent
parent	parent
parent	parstarting child process
parent	child
parent	child
ending parent process	child
starting child process	child
child	child
child	ending child process
child	ent
child	parent
child	parent
ending child process	ending parent process
$	$

Figure 4-3. Output from forktst1 Program

An Interesting Side Effect

Run the sample program, `forktst1`, again. This time redirect the output of the program to a file with the command:

```
forktst1 > testout
```

Then look at the output from the program in the file:

```
pg testout
```

The output will be like the well behaved output from before except for one major difference: the message written in the program before the **fork** call (`starting main program`) appears twice -- once before the start of each process. This repetition shows one characteristic of the **fork** call. When the main process does the **fork**, each process (parent and child) receives a copy of the buffers and open file descriptors of the parent process.

In this example, the buffer at the time of the fork contained the message, `starting main program`, and each process got a copy of that message. When the processes ended, the

system wrote each of the buffers into the same file space because both buffers contained the same file pointers. Therefore, it produces one file that contains first the complete output from one process and then the complete output from the second process. The order that the outputs occur may vary, but they are not mixed as in the previous example when the program wrote to the terminal.

Another Experiment

Copy the source file `forktst1.c` into another file called `test.c` using the command:

```
cp forktst1.c test.c
```

and then edit the new file, `test.c`. Delete the line in the child process code that contains the **exit** system call:

```
exit(0);
```

Store the file. Then compile the file, rename **a.out**, and run the new program file. This time the parent process code executes twice: once as part of the child process and once as part of the parent process. This operation produces some confusing messages because the child process is writing out *parent* messages. The **exit** call is in the original program to end the child process and prevent the child process from falling into the parent code.

Note: This experiment also shows that the child process gets a complete copy of the original parent program code, because the child does run code that it should not.

The **exit** system call also returns a completion code to the parent process if the parent uses a **wait** system call (see "Example of Fork and Wait System Calls").

Example of Fork and Wait System Calls

The program in Figure 4-4 on page 4-11 uses the **fork** and **wait** system calls in a simple program that runs. If you are using the Programming Examples, the program is stored as `forktst2.c`. Compile the program using the **cc** command and copy it to its own file, `forktst2`, as in the previous **fork** example.

To run the program, enter the command:

```
forktst2
```

When the program runs, it writes output to the screen. The two processes that the program creates both write to the screen, but this time the parent waits for the child to complete before it writes to the screen. Therefore, the result that appears on the screen is much more predictable than the previous example. The output looks like Figure 4-5 on page 4-12.

```
#include <stdio.h>
#define PR(value)  fprintf(stdout,"%s\n",(value) )
#define PD(value)  fprintf(stdout,"%d\n",(value) )

main()
{
  int c_count;
  int p_count;
  int status;
  int frkpid;
  int chldpd;

  PR("starting main process");
  PD(frkpid);
  if( (frkpid = fork() ) == 0)
  {
     PR("starting child process");
     for(c_count = 0; c_count <= 4; c_count++)
        PR("child");
     PR("ending child process");
     PD(frkpid);
     exit(0);
  }
  chldpd = wait(&status);
  PR("starting parent process");
  for(p_count = 0; p_count <= 4; p_count++)
     PR("parent");
  PR("ending parent process");
  PD(chldpd);
  PD(status);
}
```

Figure 4-4. Fork and Wait System Calls - Sample Program

```
starting main process
0
starting child process
child
child
child
child
child
ending child process
0
starting parent process
parent
parent
parent
parent
parent
ending parent process
96
0
$
```

Figure 4-5. Output from forktst2 Sample Program

Example of Exec System Call

The program in Figure 4-6 on page 4-13 uses **exec** system call together with the **fork** and **wait** system calls in a simple program that runs. If you are using the Programming Examples, the program is stored as `forktst3.c`. Compile the program using the **cc** command and copy it to its own file, `forktst3`, as in the previous **fork** example.

To run the program, enter the command:

```
forktst3
```

When the program runs, it writes output to the screen. The two processes that the program creates both write to the screen; the parent waits for the child to complete before it writes to the screen. The output looks like Figure 4-7 on page 4-14.

```
#include <stdio.h>
#define PR(value)  fprintf(stdout,"%s\n",(value) )

main()
{
  int p_count;
  int status;
  int frkpid;
  int chldpd;

  PR("starting main process");
  if( (frkpid = fork() ) == 0)
  {
     PR("starting child process");
     execl( "/bin/sh", "sh", "-c", "date",NULL);
  }
  chldpd = wait(&status);
  PR("starting parent process");
  for(p_count = 0; p_count <= 4; p_count++)
     PR("parent");
  PR("ending parent process");
}
```

Figure 4-6. Exec System Call - Sample Program

The child portion of this program does not contain an **exit** call because when it performs the **execl** call, the new program, **date** loads on top of the program in the child process (destroying any program code that is there). When the new program ends, it stops the child process and returns a completion code to the waiting parent process.

Example Pipe System Call

The program in Figure 4-8 on page 4-15 uses the **pipe** system call in a simple, but operating, program. If you are using the Programming Examples, the program is stored as `pipetst.c`. Compile the program using the **cc** command and copy it to its own file, `pipetst`, as in the previous **fork** example.

```
starting main process
starting child process
Wed May 15 15:26:25 CDT 1985
starting parent process
parent
parent
parent
parent
parent
ending parent process
$
```

Figure 4-7. Output from forktst3 Sample Program

```
#include <stdio.h>
#include <signal.h>
#define PR(value)  fprintf(stdout, "%s\n", (value) )
#define WRITE                1
#define READ                 0
#define chkmd(a, b)          (mode == READ ? (b) : (a) )

int   p_pid;

main()
{
  int w_fdesc;
  int lnmsg = 22;

  w_fdesc = openp("pr", WRITE);
  if (write(w_fdesc, "Writing piped message.", lnmsg) != lnmsg)
     PR("error writing to pipe");
  closep(w_fdesc);

}

openp(cmd,  mode)
char *cmd;
int  mode;
{
  int p_fdesc[2];

  if (pipe(p_fdesc) < 0 )
     return(NULL);
```

Figure 4-8 (Part 1 of 2). Using the pipe System Call

```
  if ( (p_pid = fork() ) == 0)
  {
     close(chkmd(p_fdesc[WRITE], p_fdesc[READ]) );
     close(chkmd(0, 1) );
     dup(chkmd(p_fdesc[READ], p_fdesc[WRITE]) );
     close(chkmd(p_fdesc[READ], p_fdesc[WRITE]) );
     execl("/bin/sh", "sh", "-c", cmd, 0);
     _exit(1);
  }
  if (p_pid == -1)
     return(NULL);
  close(chkmd(p_fdesc[READ], p_fdesc[WRITE]) );
  return(chkmd(p_fdesc[WRITE], p_fdesc[READ]) );
}

closep(fd)
int fd;
{
   register r;
   int status;

   close(fd);
   while ( (r = wait(&status) ) != p_pid && r != -1);
   if (r == -1)
      status = -1;
   return(status);
}
```

Figure 4-8 (Part 2 of 2). Using the pipe System Call

To run the program, enter the command:

```
pipetst
```

This program shows how to handle pipe file descriptors to use pipe for interprocess communication. It creates a pipe, forks another process, and writes a simple message to the pipe. The child process writes the message to the screen after receiving the message through the pipe.

When the program runs, it calls **openp** to open a pipe and fork to create a new process. The **pipe** call returns two file descriptors to the calling program: the first describes the end of the pipe to use for writing; the second describes the end of the pipe to use for reading. Both processes must cooperate to use the pipe. If one process writes to the pipe, the other process should not write to the same pipe at the same time. Similarly, if one process reads from the pipe, the other process should write something into the pipe to be read. Therefore, after the fork in this program, the child process closes the write file descriptor for the pipe and the parent process closes the read file descriptor for the pipe. The parent can then write into the pipe and the child can read from the pipe, using the remaining open file descriptors.

Note: To have two-way data transfer between processes, open two different pipes: one for reading and one for writing. Do not use one pipe for two-way data transfer.

After the child process closes the read end of the pipe, it also closes the file descriptor for standard input that belongs to that process (file descriptor 0). Then it uses a **dup** system call to copy the read file descriptor for the pipe. Because the system assigns file descriptors starting at the lowest number available and file descriptors 0 and 1 are now available, the copy of the read end of the pipe becomes standard input for the child process. The child then closes the original read file descriptor for the pipe, leaving only the file descriptor for standard input assigned to the pipe in the child process.

Having set up the pipe as standard input, the child process loads and executes (**execl**) the shell and the command that is passed to it from the parent process (**pr**). The new program in the child process then waits for input from its standard input, the read end of the pipe.

The parent process closes the read file descriptor for the pipe and returns the write file descriptor to the calling process (**pipetst**). **Pipetst** writes a message into the pipe, which is written out by the **pr** command running in the child process.

Because the **pr** command writes a formatted page output to the screen (standard output), the output may flash on the screen too fast for the user to see. If this occurs, use the command:

```
stty page length 24
```

to set the terminal to paging mode. The output fills the screen and then waits for you to press **Enter** before it displays another screen of data.

Process Identification Calls

Each process in the system has a unique number (***process ID***) that the system uses to control the activities in the system. In addition to the process ID, the system also assigns the following identifiers to a process (see Figure 4-9 on page 4-20):

Parent Process ID
The process ID of the parent process that issued a fork call to create the process.

Process Group ID
The process ID of the process that issued a fork call to create a group of processes.

TTY Group ID
A process ID that identifies all processes that started from a particular terminal.

Real User ID
The user ID of the process that started the process.

Real Group ID
The group ID of the process that started the process.

Effective User ID
The user ID that determines what files the program can access. In most cases this is the same as the real user ID; however, you can create a process that has access permission that is different from your own, and the effective user ID would be different. Refer to the **exec** system call in *AIX Operating System Technical Reference* for information about setting the set user ID bit.

Effective Group ID
The group ID that determines what files the program can access. In most cases this is the same as the real group ID; however, you can create a process that has access permission that is different from your own, and the effective group ID would be different. Refer to the **exec** system call in *AIX Operating System Technical Reference* for information about setting the set group ID bit.

A program can determine any of the above identification codes with the following system calls:

Call	Description
getegid	Gets effective group ID of the calling process.
geteuid	Gets effective user ID of the calling process.
getgid	Gets real group ID of the calling process.
getpass	Reads a password.
getpgrp	Gets process group ID.
getpid	Gets process ID.
getppid	Gets parent process ID.
getuid	Gets real and effective, user and group IDs.
setgid	Sets group ID.
setpgrp	Sets process group ID.
setuid	Sets user ID.
ulimit	Gets and sets user limits.
uname	Gets name of current system.
unamex	Gets information about current system.

See *AIX Operating System Technical Reference* for complete information about any of these system calls.

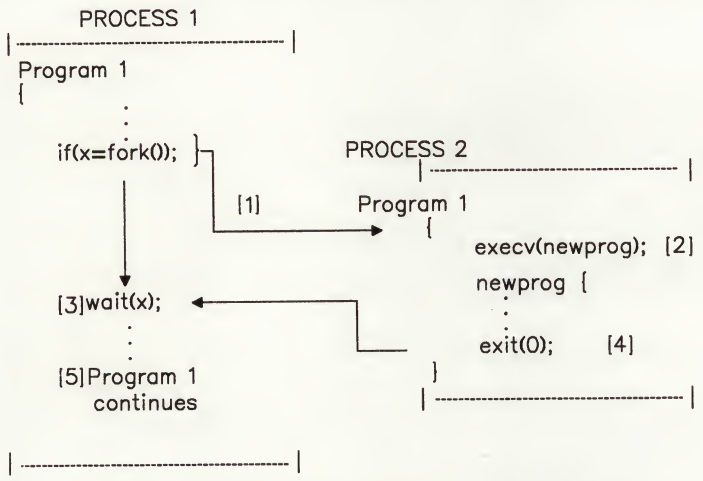


Figure 4-9. Relationship of IDs in the System

Concurrent Groups

The operating system allows you to belong to many different groups. As a group member, you can access the files whose permission bits allow access to members of that group. If you belong to more than one group, you can access the files that are available to all of those groups at any time. The ability to access files from many groups at the same time is called ***concurrent groups***.

When you log in to the system, the **login** program checks **/etc/passwd** and **/etc/group** to determine group membership. The **login** program then sets up access to the files of those groups, and assigns a group ID to indicate which of the groups is the ***primary group***. When you create files, those files become available to other members of the primary group. To change the primary group, use the **newgrp** command to change the group ID to that of another group to which you belong.

You can run a program only if you own it, if it is available to one of your groups, or if the *others* permission of the program allows you to execute it. When the program runs, it behaves much like a user with respect to groups. It normally takes on the group memberships of the person running the program. For example, if you run a program, that program can access all files that you can access; if another user runs the same program, the program no longer can access your files, but can access the files that the new user can access. You can change this behavior by using the **chmod** command to set the program to always run under its own set of user and group IDs. See *AIX Operating System Commands Reference* for information about the **chmod** command.

Using IDs with Distributed Services

When using Distributed Services, the requesting process may not be running on the node that holds the file, but the system must still check access permissions. To allow for permission checking on the remote system, the remote system translates the user ID, group ID and concurrent group IDs of an inbound request into local user IDs and group IDs. Each node has a set of mapping tables that translate the user ID or group ID of the user at the source node into a corresponding user ID or group ID at the local node. The normal sequence is:

1. The local system sends the requester's user ID, group ID and group ID list over the network along with the request.
2. The remote system uses the mapping tables to translate the IDs into their local equivalents.
3. The remote kernel process that executes the transaction program associated with the request receives the effective user ID, group ID and concurrent group list determined from the mapping tables.
4. The remote system calculates file access permissions in the normal manner based on the effective user ID and concurrent group list of the requesting process.

The mapping tables are maintained on each system by the system administrator at that node.

In addition, the administrator can set up optional outbound user ID mapping tables. If several nodes are managed by the same administrator, then that administrator can create tables for those nodes that put the same user ID on the network for a physical user no matter what the user's local user ID. Outbound translation simplifies the inbound mapping tables for those nodes managed by that administrator. A network user ID from any of those nodes always maps to a single local user ID regardless of which nodes generated the request.

Refer to *IBM RT Managing the AIX Operating System* for information about setting up inbound and outbound mapping tables for a node with Distributed Services. System calls that use the user ID or group ID understand the translation process. These system calls include:

- **chown**
- **stat**
- **fstat**
- **fullstat**
- **ffullstat.**

In addition, the **loadtbl** system call loads the translation table information into the kernel so that it can be used.

Changing the Owner of a File under Distributed Services

To allow existing programs to continue to work correctly with Distributed Services, the **chown** system call does not directly place its argument into the inode of a file. Instead, it translates the IDs that are passed to it and places translated IDs into the inode.

To place a raw, untranslated ID into the inode of a file, use the new system call **chownx**. New programs can use this system call for direct control of the values being placed into the inode. **chownx** is a more general purpose replacement for **chown** in new programs. It can be used to set either the user ID or group ID or both with translated or raw IDs.

Getting File Statistics under Distributed Services

Note: In the following discussion, ID can be either a user ID or a group ID.

The **stat** system call does not return a raw, untranslated ID from a remote file when using Distributed Services. Instead, **stat** works backwards through the translation process to determine the corresponding ID on the local system for the raw ID on the remote system. As long as only one local ID corresponds to the remote raw ID, then **stat** returns that local ID for the file. However, if the remote raw ID corresponds to many local IDs, then the following rules determine what **stat** returns as the ID:

- If the ID of the calling process is one of the many process IDs that correspond to the remote raw ID, **stat** returns the ID of the calling process.
- If the ID of the calling process is not one of the many process IDs that correspond to the remote raw ID, **stat** returns a special ID value. This value is determined by the value assigned to the variable *netsomeone* in **/etc/system** on the local system. The value of this variable is 0xFFFE if it is not changed by the system administrator.

In addition, the remote raw ID may not correspond to any ID on the local system. In that case, **stat** returns a special ID value. This value is determined by the value assigned to the variable *netnoone* in **/etc/system** on the local system. The value of this variable is `0xFFFF` if it is not changed by the system administrator.

To determine more precisely which IDs are associated with a remote file under Distributed Services, use the **fullstat** or the **ffullstat** system calls. These calls provide both the remote raw ID and the corresponding local IDs. If the local IDs are not unique, use the **loadtbl** system call to get the mapping table information for both the local and remote nodes. With this information, a program can determine all IDs that correspond to the owner or group IDs of a file.

Example of Process ID

The program in Figure 4-10 on page 4-24 uses the process ID system calls in a simple, but operational, program. If you are using the Programming Examples, the program is stored as `ppidtst.c`. Compile the program using the **cc** command and copy it to its own file, `ppidtst`, as in the previous **fork** example.

To run the program, enter the command:

```
ppidtst
```

When the program runs, it writes output to the screen. The program creates three processes: a parent, a child and a grandchild. Each process writes messages to the screen to indicate its:

- Process ID
- Process group ID
- Parent process ID
- Child process ID (except for the grandchild process).

The processes cooperate so that the output to the screen is orderly. The output looks like Figure 4-11 on page 4-25. Notice that:

- All process group IDs are the same, and are equal to the parent process ID of the parent process (the shell is the parent of the parent process in this case).
- All process IDs are different.
- The parent process ID of each of the child processes is the same as the process ID of its parent process.
- The child process ID returned to each parent is the same as the process ID of its child.

The **ps** (process status) command reports the process ID of active processes. Use the **ps** command to verify that the shell is the parent process of the parent process when you run this example.

```
#include <stdio.h>
#define PR(value)  fprintf(stdout, "%s\n", (value) )
#define PRT(s1, v1) fprintf(stdout, "%s%d.\n", (s1), (v1) )

main()
{
  int status1;
  int status2;
  int frkpid1;
  int frkpid2;
  int chldpd;
  int g_chldpd;

  PR("starting main process");
  if( (frkpid1 = fork() ) == 0)
  {
     if( (frkpid2 = fork() ) == 0)
     {
        PRT("grandchild: my process id is ", getpid() );
        PRT("grandchild: my parent process id is ", getppid() );
        PRT("grandchild: my process group id is ", getpgrp() );
        PR("ending grandchild process");
        exit(0);
     }
     g_chldpd = wait(&status2);
     PRT("child: my child's process id is ", g_chlpd);
     PRT("child: my process id is ", getpid() );
     PRT("child: my parent process id is ", getppid() );
     PRT("child: my process group id is ", getpgrp() );
     PR("ending child process");
     exit(0);
  }
```

Figure 4-10 (Part 1 of 2). Example of Process ID System Calls

```
  chldpd = wait(&status1);
  PRT("parent: my child's process id is ", chldpd);
  PRT("parent: my process id is ", getpid() );
  PRT("parent: my parent process id is ", getppid() );
  PRT("parent: my process group id is ", getpgrp() );
  PR("ending parent process");
}
```

Figure 4-10 (Part 2 of 2). Example of Process ID System Calls

```
starting main process
grandchild: my process id is 266.
grandchild: my parent process id is 265.
grandchild: my process group id is 221.
ending grandchild process
child: my child's process id is 266.
child: my process id is 265.
child: my parent process id is 264.
child: my process group id is 221.
ending child process
parent: my child's process id is 265.
parent: my process id is 264.
parent: my process group id is 221.
parent: my parent process id is 221.
ending parent process
$
```

Figure 4-11. Output for Process ID System Call Program

Changing the Controlling Terminal

Typically, each process is associated with a process group and has a controlling terminal. Each process group has a group leader process. For example, when you log in, the shell process is the process group leader and any process descendants are in that process group. The terminal you are typing on is the controlling terminal.

However, it is sometimes necessary for a program to establish the controlling terminal on its own or to disassociate itself from the process group and not have a controlling terminal. You use the **setpgrp** system call in both cases, and in both cases it is important that you perform a series of steps completely and in the correct order. If you do not follow the correct procedures, problems can occur that are sometimes intermittent and always very difficult to diagnose.

Establishing a Controlling Terminal

To establish a controlling terminal, perform the following steps in the following order:

1. Close all the file descriptors of the controlling terminal for the current process, if there are any.
2. Issue the **setpgrp** system call. This makes the current process the group leader.
3. Open the desired terminal. If this terminal is not already a controlling terminal for some other process group, it becomes the controlling terminal for this process group. The rule is as follows: The first group leader process to open a terminal that is not already a controlling terminal, acquires that terminal as a controlling terminal for that process group.
4. Issue the **dup** system call so that additional file descriptors also refer to that terminal. File descriptors 0, 1, and 2 refer to the terminal as the default.

The following program fragment illustrates these steps:

```
close(0);
close(1);
close(2);

setpgrp();

if ( open("/dev/console",O_RDWR) == -1)
  return(errno);
else
{
  dup(0);
  dup(0);
}
```

If you fail to establish the process group correctly, some functions behave improperly. For example, the **SIGINT** signal is sent to all processes in the process group and if the process group is incorrect for a particular terminal, the signal is not sent or is sent to the wrong group of processes. The **SIGHUP** signal is sent to all processes in the process group when the group leader exits and if the process group is incorrect, the signal behaves incorrectly.

Eliminating the Controlling Terminal

It is sometimes necessary that a program not have a process group and a controlling terminal. For example, a program that runs in the background might want to write error or information messages to a terminal (such as the system console) but not accept information from the terminal.

To eliminate the controlling terminal, perform the following steps in the following order:

1. Close all the file descriptors for the controlling terminal for the current process if there are any. Do this even if the inherited files are for the same terminal as the desired output terminal.
2. Open the desired output terminal.
3. Issue the **setpgrp** system call. This makes the current process the group leader. This also means that the terminal opened in the previous step is not the controlling terminal. In fact, this process group has no controlling terminal.

The following program fragment illustrates these steps:

```
close(0);
close(1);
close(2);

if ( open("/dev/console",O_RDWR) == -1)
  return(errno);

setpgrp();
```

In this situation, the program can write to the terminal (system console) but should not read from it. The interrupt key from the terminal does not affect the program because the terminal is not the controlling terminal for that process group. It is valid for another process to obtain the terminal (system console) as a controlling terminal.

A typical scenario would be where you invoke a background program from the **/etc/rc** shell and the background program, traditionally called a ***daemon***, uses **/dev/console** as an output device for messages. **/dev/console** is enabled to allow another user to log in to it.

Failure to observe the above rules could cause the following undesirable behavior for the user who managed to log in to terminal **/dev/console**:

- The interrupt key might not work as expected because the terminal is not the controlling terminal for the proper group.
- Many commands would fail because **/dev/tty** could not be opened. **/dev/tty** is a pseudo device that is the controlling terminal for some command processes. If the process has no controlling terminal, the open for **/dev/tty** fails.

Process Tracking Calls

The system provides the following calls to monitor the operation of a process or group of processes in the system. These calls are the means by which the system commands of similar names are created. Use these calls within a program to gain information about how the program runs in each section of the program. They are primarily tools for use during development which would not be included in the final version of the program. See *AIX Operating System Technical Reference* for information about the syntax and flags of these calls.

Call	Description
acct	Enables/disables process accounting.
profil	Provides an execution time profile.
ptrace	Provides a process trace.
times	Gets process and child process times.

Interprocess Communications

The system provides many methods for two or more processes to communicate with each other. It provides system calls for:

- Sending and receiving signals.
- Setting and reading semaphores
- Sending and receiving messages.

Each of these sets of calls provides one method of sending small pieces of information between processes. For larger blocks of data, use the shared memory facility described in "Shared Memory Calls" on page 4-74.

Signal Calls

Signals provide a simple method of communication between two processes. Using signals, one process can inform another process of status conditions that occur during the process, or can tell the other process when an event occurs. Use signals to activate a process for error recovery, or to help control access to shared resources (see "Semaphore Calls" on page 4-42 and "Shared Memory Calls" on page 4-74 for more advanced control of shared resources). The signals that a program can use are defined in the system file, **/usr/include/sys/signal.h** (see **signal** in *AIX Operating System Technical Reference*). The system calls that allow a program to use signals are:

Call	Description
alarm	Sets a process alarm clock.
kill	Sends signal to process(es).
pause	Suspends process until signal.
signal	Specifies what to do when the process receives a signal.
wait	Waits for child process to end.

The signal is an integer value. The meanings of these values are defined in the header file **/usr/include/sys/signal.h**. The system generates most of these signals to tell a program of error or status conditions in the system. A program should be able to respond to these conditions. The program can, however, use a few of the signals for its own purposes. Figure 4-12 lists the user signals.

Signal	Value	Definition
SIGALRM	14	Use the **alarm** system call to set a time value. The system sends **SIGALRM** to the program when that time is up.
SIGUSR1	16	A signal defined by cooperating processes to mean the same thing to those processes. One process can generate this signal using the **kill** system call. The other cooperating processes that receive the signal can then react to the condition indicated by that signal.
SIGUSR2	17	Another signal defined by cooperating processes.

Figure 4-12. User Controlled Signals

How to React to a Signal

If a program does not have code to handle signals that it could receive, it will end when it receives a signal. Therefore, include enough code to recognize all signals that the program might receive (including system error signals). Use the **signal** system call to define the actions to be performed when the program receives a particular signal. The format of the **signal** call is:

```
signal(sig,func);
```

where:

sig Is the integer for the signal, or the system macro name that represents that value (defined in **sys/signal.h**)

func Is a function code to indicate which action to take when the program receives the signal. This code can indicate to:

- End a program when the indicated signal occurs (a function code of **0**).

- Ignore the indicated signal (a function code of **1**).
- Go to a predefined routine to handle the condition (a function code that is the address of the routine).

For example:

```
signal(SIGPIPE,SIG_IGN);
```

tells the program to ignore a signal that indicates a pipe error.

Example of Trapping a Signal

The program in Figure 4-13 on page 4-33 uses the signal system call in a simple program that runs. If you are using the Programming Examples, the program is stored as `sigtst.c`. Compile the program using the **cc** command with the output going to the file, `sigtst`, as in the previous **fork** example.

To run the program, enter the command:

```
sigtst
```

When the program runs, it asks you to press any letter key (the `e` letter key to exit the program). Whatever letter you enter, the program echoes to the screen. That is the main part of the program, but it does not use signal processing.

You can generate an interrupt signal to the program (SIGINT) by pressing the key that the driver program for the terminal defines as sending that signal. This key is often the **Ctrl-Backspace** key sequence. When the program receives this signal, it branches to a routine that writes a message to the screen. The program then returns to its main program operation.

Although this function is not very useful, it shows how to build interrupt handling routines into a program. The program first tests whether interrupts (SIGINT) are ignored, and creates the jump to the interrupt trap routine only if interrupts are not ignored.

```
 /* assign the previous value of */
 /* SIGINT to istat.    */
istat = signal(SIGINT, SIG_IGN);
 .
 .
 .
/* check previous condition of SIGINT */
if(istat != SIG_IGN)
        signal(SIGINT,onintr);
        /* jump to onintr when an  */
        /* interrupt signal occurs */
```

The program makes this check because, if the program is run in background, the shell sets SIGINT to be ignored. Therefore, the system only passes the interrupt signal to foreground processes. Testing this signal condition allows the program to get the interrupt signal when running in background as well as in foreground.

The program also uses the **setjmp** and **longjmp** library functions to handle the return to the main program. Before the program begins running the main loop, it stores the machine status in a structure `sjbuf`. The structure type, `jmp_buf` is defined in the system file **setjmp.h**. When the interrupt occurs, the system transfers control to the `onintr` routine. This routine is a dummy interrupt handling routine. When it finishes, it uses the **longjmp** function to set the machine status to the conditions stored in the `sjbuf` structure. The main routine begins running again as if an interrupt had not occurred.

Each time the program returns from the `onintr` routine, it sets up the interrupt signal condition again. If it did not do this, it would not receive any more interrupt signals. The system sends only one interrupt signal for each request it receives. Once it sends that signal, the program must request another. If another interrupt signal occurs before the program sets up to catch the interrupt signal, the program stops. The enhanced signal facility (see "Enhanced Signal Facility" on page 4-34) provides a way to avoid this problem.

```
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
jmp_buf sjbuf;

main()
{
   int (*istat)(), onintr();
   int i;
   char c;
   istat = signal(SIGINT, SIG_IGN);
   setjmp(sjbuf);
   if (istat != SIG_IGN)
      signal(SIGINT, onintr);
   printf("%s\n", "Please enter a letter (e to exit).");
   while (c != 'e')
   {
      c = fgetc(stdin);
      if (c != '\n' && c != 'e')
      {
         printf("%s%c\n", "You typed the following letter: ", c);
         i++;
         printf("%d\n", i);
         printf("%s\n", "Please enter a letter (e to exit).");
      }
   }
}

onintr()
{
   printf("%s\n", "You pressed an interrupt key!");
   longjmp(sjbuf);
}
```

Figure 4-13. Example of Signal Trapping

Enhanced Signal Facility

In addition to the standard signal facilities described in "Signal Calls" on page 4-29, the system provides an enhanced signal facility. The enhanced facility treats signals received from the system in a way similar to the handling of hardware interrupts, allowing a program to mask each type of signal while it is processing. The facility allows for up to 32 different signals, but only those defined in the file **/usr/include/sys/signal.h** can be used. These are the same signals that the standard signal facility uses. You can choose to use either facility in a program, but for programs used only on the RT workstation, use the enhanced signal facility.

The system calls that comprise the enhanced signal facility are listed in Figure 4-14. *AIX Operating System Technical Reference* provides complete information for each of these calls.

Call	Function
sigblock	Sets specified bits in the signal mask to block the signals associated with those bits.
sigsetmask	Sets the signal mask to a new value.
sigpause	Sets the signal mask to a new value, waits for a signal allowed by the new mask and restores the old mask value when a signal is received.
sigstack	Defines a separate stack to contain signal processing.
sigvec	Establishes conditions to handle a specified signal.
execve	Starts a new program in the current process, resets all signals that are being caught by the old program to terminate the new program, resets signal stack state, and retains the current signal mask value.

Figure 4-14. Enhanced Signal Calls

Responding to Signals

A program can receive signals from the sources shown in Figure 4-15.

Source	Description
Program faults	A programming error such as an illegal instruction or memory reference produces this type of message.
Terminal keys	Special key sequences generate signals to running processes in the system that belong to the terminal group of the terminal. These signals include **interrupt**, **quit**, **hangup** or **kill** (the **kill** signal cannot be masked).
Hardware exceptions	Hardware faults, such as a power failure, generate signals to running processes.
System	The system can generate signals, such as the death of a child process, to aid in process control.
Other processes	Other processes can send signals to a process to coordinate system activities.

Figure 4-15. Sources of Signals

Figure 4-16 on page 4-36 shows the defined responses for responding to a signal once a program receives it. Each of these responses is defined in the header file **signal.h** as a pointer to a routine that returns the constant value assigned to that response. For example, that file defines SIG_DFL as follows:

```
#define  SIG_DFL     (int (*) ( ) ) 0
```

This definition provides a function that always returns the value 0. Defining the responses as pointers to functions allows you to use them in the **sigvec** structure (see "Using Enhanced Signals" on page 4-36) as the pointer to the handler routine. See "Example Programs" on page 4-40 for a sample of using the responses in the structure.

Response	Meaning
SIG_DFL	Specifies that the system should perform the default action for a specific signal when it receives that signal. For most signals, the default action is to terminate the process. For some signals the default action ignores the signal, or creates a core file before terminating.
SIG_IGN	Specifies that the system should ignore the specified signal when it occurs.
SIG_CATCH	Specifies that the system should call a procedure in the current process when the specified signal occurs. The system also blocks any other occurrences of the same signal while the process is handling the signal.
SIG_HOLD	Specifies that the system should block a specified signal when it occurs. When blocked, the signal remains pending until the block is released. The signal is not lost.

Figure 4-16. Signal Responses

Using Enhanced Signals

When a process receives a signal, the system automatically:

1. Blocks another signal of that type from being sent to the process.
2. Saves the environment of the process.
3. Builds a new environment for the process to respond to the signal.
4. Transfers control to the signal ***handler*** routine in the process.

The signal ***handler*** is a routine that you provide to respond to the receipt of a signal. It may be a complex error recovery routine, or it may be SIG_DFL. You choose how the program responds to each of the signals. Once you have created the handler routines for all of the signals that you expect to receive, tell the system how to handle each of the signals. First, include the header file **signal.h** with the following statement:

```
#include <signal.h>
```

This file contains definitions for all the constant names used by the signal handling facility. In addition, this file contains the following structure definition:

```
struct  sigvec
{
        int     (*sv_handler) ( );
        int     sv_mask;
        int     sv_onstack;
};
```

This is the structure type that passes information to the system when using the **sigvec** system call. It contains three integer members as shown in Figure 4-17 on page 4-37.

Name	Description
(*sv_handler) ();	A pointer to the routine that you have created to handle the signal processing for a particular signal, or one of the responses defined in **signal.h** as pointers to routines that return constant values.
sv_mask;	A 32-bit mask containing one bit for each of the 32 possible signals (not all are used). If a bit in this mask is set, then the system adds the corresponding signal to the current set of signals that are blocked from interrupting the handler routine while the handler is running. In addition, the signal that caused the handler to be run is also blocked.
sv_onstack;	Only bits 0 and 1 of this integer are valid; the rest of the bits should be zero. Valid values for this integer are: **0** Use enhanced signals and process signals on the process stack. **1** Use enhanced signals and process signals on a separate stack that you specify using the **sigstack** system call. **2** Use standard signal processing **3** Not a valid value.

Figure 4-17. sigvec Structure Members

For example, to set up a routine to receive the SIGALRM signal, use the following system call:

```
sigvec( SIGALRM , new_vec , old_vec );
```

The parameters for this call have the following meaning:

Parameter Meaning

SIGALRM The alarm clock (signal 14).

new_vec A pointer to the structure of type `sigvec` that defines the information for the signal handler.

old_vec A pointer to a structure of type `sigvec` where the system returns the values it is using for the specified signal. If this pointer is `NULL`, the system ignores it.

Using a Separate Signal Stack

You can choose to handle signal conditions using a stack that is separate from the process stack. Using a separate stack is not often useful or needed. Use it when catching the signal SIGSEGV to detect a stack overflow. You can then use the small signal stack to operate from while trying to recover from the overflow condition. However, when using a separate stack, you must control all stack operations and detect overflow. The system does not provide control for the separate stack. To use a separate stack for a particular handler, or set of handlers, perform the following steps:

1. Use the **sigstack** call to create a signal stack for handlers in the process to use.
2. Set the value of the **sv_onstack** variable to `1` in the structure of type `sigvec` for each of the handlers that use the separate stack.

Waiting for a Signal

The **sigpause** system call stops processing in a program to wait for the occurrence of any or all of the signals, and then resume processing without altering the signal mask (`sv_mask`) with which the program normally operates. The following sequence illustrates how to stop program operation, wait for any signal to occur, and then resume program operation:

1. Program begins.
2. Program issues **sigpause** call with a mask value of zero to enable catching all signals:

   ```
   sigpause( 0 );
   ```

3. A signal occurs for the process.
4. The **sigpause** call returns with a `-1` return code, and **errno** set to EINTR.
5. The system restores the previous signal mask value.
6. The program continues.

Protecting Important Program Events

During some parts of a program, you may want to continue processing in spite of any signals that the program may receive. Some activities, such as processing a linked list, could be difficult or impossible to recover from if the program were interrupted at that point. To protect sections of the program from interruption from all but the most serious signals (SIGKILL cannot be masked), use the **sigblock** and **sigsetmask** system calls together, as shown in the following sequence:

1. At the beginning of the important section of the program, use the **sigblock** system call with a mask value that, when ORed with the current mask, blocks all expected signals. Provide an integer variable to hold the returned value (the old mask value) for later use:

   ```
   old_mask = sigblock( block_mask );
   ```

2. Process the important section of the program.
3. Restore the program to normal operation with the **sigsetmask** system call, using the saved mask value from the **sigblock** call:

   ```
   block_mask = sigsetmask( old_mask );
   ```

4. Continue normal program operation.

Finding Out the Current Signal Mask

Each process running in the system has its own signal mask, regardless of whether or not the process uses signals. If the program does not use the signal calls to change this mask, the system assigns a value of zero to the signal mask (no signals blocked).

Two system calls provide information about the signal conditions that a program is using for a specified signal. The first call, **sigvec**, requires a defined structure of type **sigvec** where the system can write the current information. The following call tells the system to store the current information for the signal SIGINT in the structure `old_vec`. Because the second parameter that normally points to the new signal structure is NULL, the current signal structure for SIGINT does not change, but it is copied to the structure `old_vec`.

```
sigvec( SIGINT , NULL , old_vec );
```

You can use the **sigblock** system call to get only the value of the current mask:

```
old_mask = sigblock( 0 );
```

This call ORs the supplied value of zero with the current mask and returns the value of the current mask to `old_mask`. The supplied value of zero does not change the current mask.

Example Programs

Figure 4-18 shows an example program using enhanced signals. This program performs the same function as the example program used for signals in "Example of Trapping a Signal" on page 4-31. Refer to the program description in that section. This program is stored as **newsig.c** in the example program directory.

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
char c;
int i;
jmp_buf sjbuf;

main()
{
   int onintr();
   char fgetc();
   struct sigvec oldvec, newvec;
   newvec.sv_handler = SIG_IGN;
   newvec.sv_mask = 0;
   newvec.sv_onstack = 0;
   sigvec( SIGINT, &newvec, &oldvec );
   if( oldvec.sv_handler != SIG_IGN )
   {
      newvec.sv_handler = onintr;
      sigvec( SIGINT, &newvec, NULL );
   }
```

Figure 4-18 (Part 1 of 2). Enhanced Signals Example Program

```
   setjmp( sjbuf );
   printf( "%s\n", "Please enter a letter (e to exit)." );
   while ( c != 'e' )
   {
      c = fgetc( stdin );
      if( c != '\n' && c != 'e' )
      {
        printf( "%s%c\n", "You typed the following letter: ", c );
        i++;
        printf( "%d\n", i );
        printf( "%s\n", "Please enter a letter (e to exit)." );
      }
   }
}          /* End Main */


onintr()
{
   printf( "%s\n", "Performing processing upon interrupt." );
   printf( "%s\n", "Finished interrupt processing; returning to main." );
   longjmp( sjbuf, 1 );
}                 /* End onintr */
```

Figure 4-18 (Part 2 of 2). Enhanced Signals Example Program

Semaphore Calls

Semaphores provide a general method of communication between two processes that is an extension of the features of signals. Use semaphores in much the same way as signals, except that semaphores are:

More flexible: Processes can define a semaphore to mean what they want it to mean.

More controllable: Programs have direct control over semaphores and do not need to depend on the system to generate them.

Broader in scope: A semaphore can be any integer value, not just 1 or 0. Use them for counting as well as process coordination (see program example).

The system calls that allow a program to use semaphores are:

Call	Description
semctl	Semaphore control operations
semget	Get set of semaphores
semop	Semaphore operations

Structure of a Semaphore Set

When using the **semget** system call to create a set of semaphores, the system returns an integer that is the ***semid***, or semaphore ID, for the set of semaphores that were created. Each semid points to a set of semaphores and a data structure that contains information about the semaphores. The data structure for semid is shown in Figure 4-19 on page 4-43. See Figure 4-20 on page 4-43 for the data structure of a semaphore.

Name	Function
sem_perm cuid cgid uid gid mode	Operation Permission Struct creator user ID creator group ID user ID group ID read and alter permission
sem_nsems	Number of Semaphores in the Set
sem_otime	Time of the Last Operation (seconds since 1/1/1970)
sem_ctime	Time of the Last Change (seconds since 1/1/1970)

A5AC6018

Figure 4-19. Semid Data Structure

Name	Function
semval	Value of the semaphore (0 or positive)
sempid	Process ID of the Last Operation
semncnt	The number of processes that are waiting for semval to be > current value of its last semop() call
semzcnt	The number of processes that are waiting for semval to be = 0.

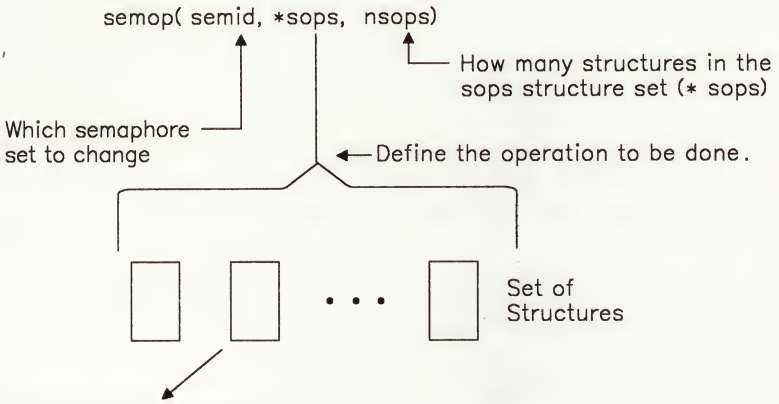
A5AC6017

Figure 4-20. Semaphore Structure

How to Use Semaphores

Semaphores are counters that a program can test and change with a single system call (**semop**). Specify the amount of the change in the **semop** call using the **sem_op** variable within the **sops** structure as shown in Figure 4-21. When using this call, the system tests the value of **sem_op** against the value of the semaphore indicated by **sem_num**. If (sem_flg & IPC_NOWAIT) is true, the call returns without further action. If it is false, the table in Figure 4-22 on page 4-45 summarizes the actions that occur.

Use semaphores for passing data between processes and for other one time data transfers. You can also use them to control access to a limited resource, such as a shared buffer.



Name	Function
sem_num	Number of the semaphore to be changed
sem_op	The semaphore operation
sem_flg	Operation flags

A5AC6016

Figure 4-21. Semop System Call Parameters

Value of sem_op	Relationship to semval	Actions
< 0	\|sem_op\| <= semval	1. semval = semval - \|sem_op\|
	\|sem_op\| > semval	1. semncnt = semncnt + 1 2. wait for semval >= \|sem_op\|; then semval = semval - \|sem_op\| semncnt = semncnt - 1
> 0		1. semval = semval + sem_op
0	semval == 0	1. return
	semval != 0	1. semzcnt = semzcnt + 1 2. wait for semval = 0; then semzcnt = semzcnt - 1 return

A5AC6015

Figure 4-22. How sem_op Specifies a Semaphore Operation

Example of Semaphores

For example, two processes `proca` and `procb` share a buffer `bufr`. `proca` produces data packages and places each package in the buffer as it produces that package. `procb` uses the data packages from the buffer in its operation, but does not use them at the same rate that `proca` puts them in the buffer. The buffer can hold only three data packages. The processes use two semaphores to ensure that:

- `proca` does not try to put a package in a full buffer.
- `procb` does not try to take a package from an empty buffer.

The processes agree that the semaphores have the following meanings:

sem1 The number of empty slots in the buffer. If sem1 is greater than zero, then `proca` can put a package in the buffer.

sem2 The number of data packages in the buffer. If sem2 is greater than zero, then `procb` can take a package from the buffer.

Figure 4-23 shows the relation of the processes to the buffer and semphores. When the processes start, `sem1` has a value of 3 (empty slots) and `sem2` has a value of 0 (packages in `bufr`).

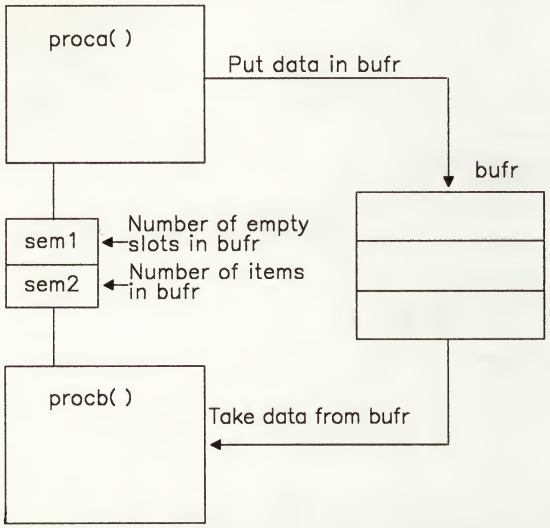


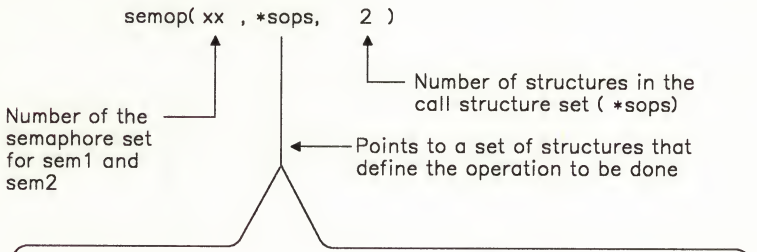
Figure 4-23. Using Semaphores Concept Example

proca

proca puts items into the buffer. Before it can put an item in the buffer, it tests seml to find out if there is room in the buffer for the item. If seml is greater than, or equal to 1 (the number of items to be put into bufr), then it:

1. Decrements seml by 1
2. Puts an item in the buffer
3. Increments sem2 by 1.

If seml is less than 1, proca waits until there is room in the buffer (seml > = 1), and then performs the preceding steps. If proca has to wait, it increments the semncnt flag for seml to indicate that it is waiting. It decrements this flag when it continues. Figure 4-24 shows the **semop** call and structure that handles all of the semaphore operations for proca.



Value	Function
1	sem2
-1	Test/decrement
0	Enable wait

Value	Function
2	sem1
1	Increment
0	Enable wait

A5AC6014

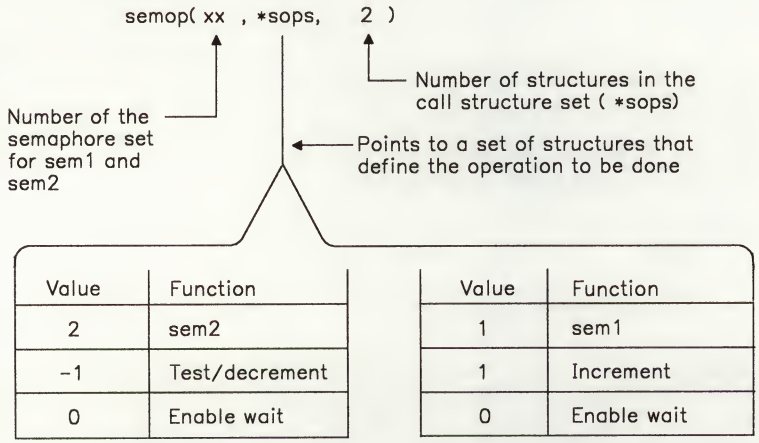
Figure 4-24. Semop Call for Proca

procb

procb takes items from the buffer. Before it can take an item, it tests sem2 to find out if there is anything in the buffer. If sem2 is greater than, or equal to 1 (the number of items to be taken), then it:

1. Decrements sem2 by 1
2. Takes an item from the buffer
3. Increments sem1 by 1.

If sem2 is less than 1, procb waits until there is something in the buffer (sem2 > = 1), and then performs the preceding steps. If procb has to wait, it increments the semncnt flag for sem2 to indicate that it is waiting. It decrements this flag when it continues. Figure 4-25 shows the **semop** call and structure that handles all of the semaphore operations for procb.



A5AC6012

Figure 4-25. Semop Call for procb

Operation

The chart in Figure 4-26 on page 4-49 shows the operation of the semaphores to control access to the buffer.

Event	sem1		sem2	
	Value	semncnt	Value	semncnt
1. Start: The initial state of the parameters	3	0	0	0
2. Procb tries to get item:	3	0	0	1
3. Proca puts item in bufr.	2	0	1	1
4. Procb can now get item.	3	0	0	0
5. Proca puts item in bufr.	2	0	1	0
6. Proca puts item in bufr.	1	0	2	0
7. Proca puts item in bufr.	0	0	3	0
8. Proca tries to put item in bufr.	0	1	3	0
9. Procb gets item from bufr.	1	1	2	0
10. Proca can now put item in bufr.	0	0	3	0

A5AC6011

Figure 4-26. Semaphore Usage

Example of Semaphore Programming

The program in Figure 4-27 on page 4-50 shows the use of semaphores in a situation similar to the example concept described previously. This program also uses some shared memory calls as described in "Shared Memory Calls" on page 4-74. If you are using the Programming Examples, this program is stored under the name **semtst.c**. You can compile and run this program to see the effects of the system calls.

```
#define STKSZ                   2
#define SNSLTS                  0
#define SNITMS                  1
static int sp = 0;
static char *itmstk[STKSZ];

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <errno.h>
#define MADDR                   0

int semid;
int shmid;
extern int errno;
struct ITM
  {
     char *itmstk[2];
     int sp;
  }  *stkptr;
```

Figure 4-27 (Part 1 of 5). Using Semaphore Calls

```
main()
{
  char *popitm();
  int nsems, semflg;
  int pid_1, pid_2;
  int stts_1, stts_2;
  register rt_1, rt_2;
  key_t key, ftok();
  char *shmat();
  int shmdt();
  struct sembuf soa1[1], soa2[1];
  struct sembuf *sops1 = soa1,
                *sops2 =  soa2;

  key = ftok( "semtst.c", 's' );
  nsems = 2;
  semid = semget( key, nsems, IPC_CREAT | SEM_A | SEM_R );
  semctl( semid, 0, SETVAL, 2 );
  semctl( semid, 1, SETVAL, 0 );
  shmid = shmget( key, STKSZ, IPC_CREAT | SHM_W | SHM_R );
  stkptr = (( struct ITM * ) shmat( shmid, MADDR,
                                    IPC_CREAT | SHM_W | SHM_R ));
  if(( pid_1 = fork() ) == 0 )
  {
     int i;

     stkptr->sp = 0;
     for( i = 0;i <= 5;i++ )
     {
        printf( "\nProducer: Sending item number %d   ", i );
        p( SNSLTS, sops1 );
        pushitm( "Test Item", stkptr );
        v( SNITMS, sops2 );
     }
     exit(0);
  }
```

Figure 4-27 (Part 2 of 5). Using Semaphore Calls

```
  if(( pid_2 = fork() ) == 0 )
  {
     int i;
     char *itmptr;

     for( i = 0;i <= 5;i++ )
     {
        p( SNITMS, sops1 );
        semctl( semid, 1, GETVAL );
        itmptr = popitm( stkptr );
        v( SNSLTS, sops2 );
        printf( "\nConsumer: Got item number %d; item is '%s'", i, itmptr );
     }
     exit(0);
  }
  while(( rt_1 = wait( &stts_1 ) != pid_1 ) &&
        ( rt_1 != -1 )) &&
       (( rt_2 = wait( &stts_2 ) != pid_2 ) &&
        ( rt_2 != -1 )))
  ;
  if( rt_1 == -1 )
  {
     stts_1 = -1;
  }
  if( rt_2 == -1 )
  {
     stts_2 = -1;
  }
  printf( "Producer process ended with status = %d\n", stts_1 );
  printf( "Consumer process ended with status = %d\n", stts_2 );
}
```

Figure 4-27 (Part 3 of 5). Using Semaphore Calls

```
int stpop, stvop;

p( s, sops )
int s;
struct sembuf *sops;
{
  int nsops;

  nsops = 1;
  sops->sem_num = s;
  sops->sem_flg = ~IPC_NOWAIT;
  sops->sem_op = -1;
  stpop = semop( semid, sops, nsops );
  return;
}

v( s, sops )
int s;
struct sembuf *sops;
{
  int nsops;

  nsops = 1;
  sops->sem_num = s;
  sops->sem_flg = ~IPC_NOWAIT;
  sops->sem_op = 1;
  stvop = semop( semid, sops, nsops );
  return;
}
```

Figure 4-27 (Part 4 of 5). Using Semaphore Calls

```
char *popitm( ip )
struct ITM *ip;
{
  if( ip->sp > 0 )
  {
     printf( "\npop: stack pointer = %d   ", --ip->sp );
     return( ip->itmstk[ip->sp] );
  }
 else
     return( NULL );
}

int s;
pushitm( item, ip )
char *item;
struct ITM *ip;
{
  ip->itmstk[ip->sp++] = item;
  printf( "\npush: stack pointer = %d   \n", ip->sp );
}
```

Figure 4-27 (Part 5 of 5). Using Semaphore Calls

Message Calls

Messages provide a general method of communication between two processes. One process can pass information of any kind to another process by sending messages. The information might be data produced by one process and used in another, or it could be flags that indicate when events occur. To use the message process, perform the following steps:

1. Use one of the following subroutines to get a key assigned to a message queue:
 - **ftok** for a normal file system only
 - **create_ipc_prof** for either a normal file system or a system that uses Distributed Services.
2. Use the **msgget** call to get a message queue assigned to the processes.
3. Use the **msgsnd** call to send a message to a queue that is assigned to another process.
4. Use the **msgrcv** or **msgxrcv** call to receive a message from the message queue.

Use the following system calls to create and use message queues:

Call	Description
msgctl	Gets status, changes permissions, or removes a queue.
msgget	Gets message queue.
msgrcv	Receives a message.
msgsnd	Sends a message.
msgxrcv	Receives a message with additional information.

In addition, the **ftok** or **create_ipc_prof** subroutines provide the key that the **msgget** call uses to create the message queue. For an application program to operate in all AIX Operating System environments, it should use **create_ipc_prof** to get the key value. The **ftok** subroutine key should not be used when Distributed Services is installed. That subroutine returns a key that is much more likely to match a value that is already in use than the **create_ipc_prof** subroutine. Include the following header files when using message queue calls:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

Terms

The use of message queue elements is similar in structure to the way the system creates and uses files. Defining the terms used for message queues with respect to the more familiar file terms provides a framework to build an understanding of message queues. Figure 4-28 shows the new terms and how they relate to terms used with files. The first three terms (key, msqid and permissions) pertain to how the kernel handles queues. The last term (queue name) is used in profiles.

Term	Definition
key	The key is a unique identifier (of type **key_t**) that names the particular message queue. It is always associated with the message queue as long as the message queue exists. In this respect, it is similar to the *filename* of a file.
msqid	The msqid is an identifier assigned to the message queue for use within a particular process. It is similar in use to a *file descriptor* of a file.
permissions	The message queue structure also contains information that describes the access permissions for the message queue. These permissions are similar in function to the access permission bits for a file (owner, group and others).
queue name	The queue name is a 14-character, alphanumeric name that applies to a specific queue. The programmer or administrator chooses the name so that it does not conflict with other queue names on the system. Programs use the queue name to create or access the queue and get a key for that queue. Subsequent operations use the key value to refer to the queue.

Figure 4-28. Message Queue Terms

In effect, message queues are a more general form of the **pipe** system call. Either method passes information between two processes. For message queues, however, you do not need to perform the steps of opening a pipe, forking, and then closing two of the ends of the pipe as described in "Example Pipe System Call" on page 4-13. In fact, the two processes using message queues to communicate do not need to be created from the same ancestor process; they only need to cooperate by using the same name for the queue, and agreeing about what the messages mean.

General Operation

When a process gets a message queue, it uses an internal name (***key***) to apply to the queue. Any other programs that use that key can access that queue, subject to the read/write access permissions set up for the queue. The process can either send messages to the queue or receive messages from the queue (or both if it wants to send messages to itself).

After opening the queue, the process continues operating until it reaches the point that it needs input from the other cooperating process. The first process checks its message queue using the **msgrcv** call. Using the parameter, ***msgtype***, in the **msgrcv** call, the process can specify which type of message it wants to receive. If a message that satisfies the request is not in the queue, the first process halts until something is put into the queue that does satisfy the request. If there is a message of the requested type in the queue, the system gives the process the first message of that type that was put into the queue (first-in-first-out).

Similarly, after opening the queue, the other process can send messages to the queue of the first process. If the queue is full, the system returns an error indication and the process must wait until the first process empties the queue enough to add the new message.

Because either of these wait conditions could halt the process indefinitely, the program should include a timeout loop to end the stalled condition.

Sending messages to a queue is completely independent from receiving messages from that queue. The amount of data that one process can put into the message queue of the other process depends on the queue size and the speed that the other process takes the data from the queue. More than one process can put messages into a queue. The receiving process must take them out of the queue in the order that they were put into the queue, modified only by selecting a message type.

The receiving process can also use the **msgxrcv** call instead of the **msgrcv** call to get messages. This call provides more information to the receiving process about the nature of the message.

Controlling Bidirectional Queues

When using a queue for a system service daemon, the requesting process can send a message to that queue and wait for a reply. In many cases, these queues can be set up for bidirectional communication; that is, both the daemon and the requesting process can get messages from the same queue. A bidirectional queue saves the overhead associated with creating and deleting a queue for the requesting process while it waits for the reply.

Use of a bidirectional queue requires cooperation among the processes. Routing the reply messages to the correct requesting process requires that each process generate and use a unique *mtype* parameter (message type) with all its messages to that queue. When sending messages to the queue, requesting processes send messages to the *mtype* of the daemon program. In each message they specify the *mtype* to which the daemon program must send the reply. Since the daemon program is set up to receive messages of all types from the queue, it gets all the messages. The daemon responds by generating a message of a type corresponding to the process for which the message is intended; the requesting process specifies in its **msgrcv** call that it will receive messages only of its unique type, and only receives the reply from the daemon that was intended for it.

To help assign unique *mtype* values, each message queue header includes a 32-bit value that contains the most recently used *mtype* value. A new command for **msgctl**, IPC_MTYP, returns the current value of the *mtype* of a queue. The *mtype* of the queue is incremented after it is returned to a process, but is not allowed to become negative.

Use the following guidelines when setting up bidirectional queues that use *mtype* for routing messages:

- Reserve the *mtype* value of 0 for requests to the daemon program.
- The requesting program uses **msgctl** with a command type of IPC_MTYP to get an *mtype* value from the queue header when the program gets a message queue ID. The kernel returns a unique *mtype* for each **msgctl** call.
- The requesting program includes its *mtype* value as part of the data for each request sent to the queue.
- When the daemon program replies, it uses the *mtype* value sent by the requesting program as the *mtype* of the reply message.
- The requesting program waits for its reply by issuing a **msgrcv** that specifies only a message type that matches the *mtype* value sent to the queue.

IPC Message Queues in a Distributed Services Environment

When using message queues in a Distributed Services environment, a program can send and receive messages through a queue that is located at a different node in the network. A process at one node can communicate with processes in other nodes using the same message queue features that work with local message queues. The actual location of the queue does not matter to the sending or receiving processes. The two processes may be in the same or different nodes without changing the programming interface. When the queue is in a remote node, the two kernels use a communications network to pass queue element information between the requesting node and the queue node.

When using queues in a Distributed Services environment, the administrator must define keys and the location (local or remote) of their associated queue. If the location is remote, the node ID of the remote node and the associated key from the remote node's key name space is also required. A 14-character queue name can be associated with a key for external references.

Selecting a Key

When selecting a key to identify a queue, ensure that the keys for queues being used by system service programs (such as print spoolers) are unique. For queues running without Distributed Services installed, the **ftok** subroutine returns a key that satisfies requirements. However, the key that **ftok** returns is not adequate for queues used with Distributed Services. It does not provide enough choices. When using a queue with Distributed Services, use the **create_ipc_prof** subroutine to get a key. This library routine creates a profile. Your program specifies the name, and the routine creates a queue profile, assigns a local key to it, and returns the local key to the program.

Note: Queue names that begin with the letters IBM are reserved for system usage.

Choose the name carefully to avoid duplicating the name chosen by another application that may be running on the system. The 14-character name allows you to carefully choose a name that is likely to be different from those chosen by other application programs. However, to be certain, you may want to incorporate backup measures into your program in case the chosen name is already in use. These backup measures can include the following:

1. Code the preferred queue name into the application program, but allow for the administrator to change the queue name by using a command flag for all programs that use the queue.
2. In the installation program for your application program, use the **create_ipc_prof** subroutine to create a queue for the profile. If the create fails because of a name duplication, then prompt the administrator to enter a new queue name. Then use the new name to create the profile.
3. If the queue name is changed, ensure that the administrator knows (through documentation or online information) that the related programs must always be started

with the queue name flag to reference the new name of the queue. To ensure correct operation, your installation program could enter the command (with the proper flag) into **/etc/rc** to start any daemon programs, and could create a small shell script that contains the command with the proper flag to replace the actual command file.

Distributed Services Queue Example

The following paragraphs describe the installation and operation of a program that uses message queues with Distributed Services. The program consists of a daemon process and a command program that makes requests to the daemon. The daemon runs on network node **A**. The command program can be run on any node in the network (including node **A**).

At node **A**, the installation program uses **create_ipc_prof** to create a profile for the needed queue. The program provides the queue name of `ACMEsfwr032686`. The created profile contains the following information:

```
Queue Name                  ACMEsfwr032686

Local Key                   0x3FFFF

Remote Key                  0

Remote Node ID/Nickname     (null)
```

The value in the Local Key field is a value in the range 0x30000 to 0xFFFFF that does not conflict with any other profile on the local node. The subroutine chooses the value.

At each node where the command program can be run, the installation program uses **create_ipc_prof** to create a profile for the needed queue. Again, the installation program provides the queue name of `ACMEsfwr032686`. The created profile contains information similar to that shown in the preceding pop-up panel, except that the Local Key field is different.

The administrator then uses the **ipctable** command to modify the profiles at the nodes where the command program runs. The administrator enters values for the following parameters. See *IBM RT Managing the AIX Operating System* for information about modifying profiles.

Remote Node ID/Nickname
The node ID or nickname that identifies the node where the daemon program runs.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
#include <drs.h>
#define MSGSIZ          15      /* common size                      */
#define MSGRTYP          0      /* receive first message on queue */
#define MTYPE            1      /* common message type              */

static char *msg[11] = {
                         "this is item 0", "this is item 1",
                         "this is item 2", "this is item 3",
                         "this is item 4", "this is item 5",
                         "this is item 6", "this is item 7",
                         "this is item 8", "this is item 9",
                         "this is item 10"
                       };
int  msgqid;
extern int errno;

main()
{
  int pid_1, pid_2;
  key_t key, rkey;
  char nid[15];
  char *receive();
  int send();
  register rt_1, rt_2;
  int stts_1, stts_2;
  int rtn;
```

Figure 4-29 (Part 1 of 4). Example of Using Message Queues

```
if( find_ipc_prof( "msgqtst.c", &key, &rkey, nid ) < 0)
{
    if( (rtn = create_ipc_prof( "msgqtst.c", &key, &rkey, nid )) < 0)
    {
         printf( "Could not create ipc profile; Error=%d\n", rtn );
         exit(0);
    }
    printf( "Created a message queue with queue name = msgqtst.c.\n" );
}
else
{
    printf( "A message queue with queue name = msgqtst.c already exists.\n" );
}
if( (msgqid = msgget( key, IPC_CREAT | MSG_R | MSG_W )) < 0)
{
     perror( "msgqtst cannot open message queue" );
     exit(0);
}
printf( "msgqid = %d\n", msgqid );
if(( pid_1 = fork() ) == 0 )
{                                /* Begin Child 1 Process */
   int i ;

   printf( "Starting producer process (Child 1).\n" );
   for( i = 0;i <= 10;i++ )
   {
      printf( "Producer process: sending message number %d\n", i );
      send(msg[i]);       /* produce item */
   }
   exit(0);
}                               /* End Child 1 Process */
```

Figure 4-29 (Part 2 of 4). Example of Using Message Queues

Remote Key

The key value used at the remote node (where the daemon program runs) to identify the queue (from the Local Key field of the profile on the daemon's node).

When the daemon starts, it uses **find_ipc_prof** with the queue name of ACMEsfwr032686 to find the key associated with that name. It then uses **msgget(** *key*, **IPC_CREATE**) to create the queue.

When a command program runs, it uses **find_ipc_prof** with the queue name of ACMEsfwr032686 to get the key associated with the queue. It then uses **msgget(** *key*, **0**) to get an ID for the queue.

Using Message Queues

The following sequence shows how to create and use a message queue:

1. Create a key to uniquely identify the message queue. Use the **create_ipc_prof** subroutine to create the key. For example, to create a key `mykey` using a project ID of `X` contained in the **char** variable `proj` and a file name of `null_file`, use a statement like:

   ```
   mykey = create_ipc_prof( null_file, proj );
   ```

2. Either:

 - Create a new message queue with the **msgget** system call. For example, to create a message queue and assign the **msqid** to an integer variable `msg_qid`, use a statement like:

     ```
     msg_qid = msgget( mykey, IPC_CREAT );
     ```

 or

 - Get a previously created message queue with the **msgget** system call. For example, to get a message queue that is already associated with the key `mykey` and assign the **msqid** to an integer variable `msg_qid`, use a statement like:

     ```
     msg_qid = msgget( mykey, IPC_ACCESS );
     ```

3. Use the queue to send or receive messages with other processes.

4. If the queue is no longer needed, eliminate it from the system using the **msgctl** system call:

   ```
   msgctl( msg_qid, IPC_RMID );
   ```

See *AIX Operating System Technical Reference* for specific information about parameters for the calls and subroutines.

Example of Message Queue Calls

The program in Figure 4-29 on page 4-63 shows the use of message queues in a simple producer-consumer relationship. One process produces an item for the other process and passes it to the other process on a message queue. See "Distributed Services Message Queue Example" on page 4-100 for a version of this program that runs with Distributed Services.

If you are using the Programming Examples, this program is stored under the name **msgqtst.c**. You can compile and run this program to see the effects of the system calls with the following command:

```
cc -o msgqtst msgqtst.c -lipc
```

```
  if(( pid_2 = fork() ) == 0 )
  {                                /* Begin Child 2 Process */
     int i;

     printf( "Starting consumer process (Child 2).\n" );
     for( i = 0;i <= 10;i++ )
     {
         printf( "Consumer process: message number %d is: %s\n",
                   i, receive() ); /* consume item */
     }
     exit(0);
  }                                /* End Child 2 Process */
  while((( rt_1 = wait( &stts_1 ) != pid_1 ) &&
         ( rt_1 != -1 )) &&
        (( rt_2 = wait( &stts_2 ) != pid_2 ) &&
         ( rt_2 != -1 )))
  ;
  if( rt_1 == -1 )
  {
     stts_1 = -1;
  }
  if( rt_2 == -1 )
  {
     stts_2 = -1;
  }
  printf( "Producer process ended with status = %d\n", stts_1 );
  printf( "Consumer process ended with status = %d\n", stts_2 );
  del_ipc_prof("msgqtst.c",&key,&rkey,nid);
  (void) msgctl (msgqid, IPC_RMID, NULL);
}   /* End Parent Process */
```

Figure 4-29 (Part 3 of 4). Example of Using Message Queues

```
struct msgbuf msgb, *msgp = &msgb;   /* define pointer to message buffer */
char *strcpy();

send( item )
char *item;
{
  msgp->mtype = MTYPE;
  strcpy(( msgp->mtext ), item );
  msgsnd( msgqid, msgp, MSGSIZ, ~IPC_NOWAIT );
  return;
}

char *receive()
{
  msgrcv( msgqid, msgp, MSGSIZ, MSGRTYP, ~IPC_NOWAIT );
  return(( msgp->mtext ));
}
```

Figure 4-29 (Part 4 of 4). Example of Using Message Queues

System Memory Management

The system provides a continuous range of memory addresses for accessing data from 0x0000000000 to 0xFFFFFFFFFF, for a total addressable space of more than 10^{12} bytes. The system reserves some ranges of addresses for use by the VRM and the input/output bus, but it treats most of the addresses as memory that the operating system can use. The actual physical memory on any system is much smaller than the address space. Because the address space does not correspond, one-to-one, with ***real memory***, the address space and the way the system makes it correspond to real memory is called ***virtual memory***. Other terms needed to define memory management are:

page A unit of virtual memory space that holds 2K bytes of data.

segment A unit of virtual memory space that holds up to 256M bytes, or 128K pages of data.

K-byte A unit of data capacity equal to 1024 bytes.

M-byte A unit of data capacity equal to 1,048,576 bytes.

To accommodate the large virtual memory space with a limited real memory space, the system uses real memory as a work space and keeps the inactive data and programs that are not mapped in an area on disk called the ***paging space***. When it needs data or a program that is in the paging space, the system:

1. Finds an area of memory that is not currently active.
2. Ensures that an up-to-date copy of the data or program from that area of memory is in the paging space on disk.
3. Reads the new program or data from the paging space on disk into the newly freed area of memory.

The process of moving data between memory and disk as the data is needed is called ***paging***. The system also has methods for tracking the most frequently used data in the paging space to shorten the time that it takes to find that data. In addition, many important pieces of code, including much of the operating system kernel, are fixed in memory and cannot be paged to disk.

The 10^{12} byte address space available implies that a program can use 40-bit (or 10 hexadecimal digits) addresses. However, programs can only use 32-bit (or 8 hexadecimal digits) addresses. To bridge the gap between the two addresses, the system uses a set of 12-bit (or 3 hexadecimal digits) ***segment registers***. The system loads these registers with values that become the high-order 12 bits of the 40-bit address needed to select all available addresses. The high-order 4 bits of the 32-bit address select which of the segment registers to use. This process, in effect, exchanges the top 4 bits for 12 bits, creating the 40-bit address as shown in Figure 4-30.

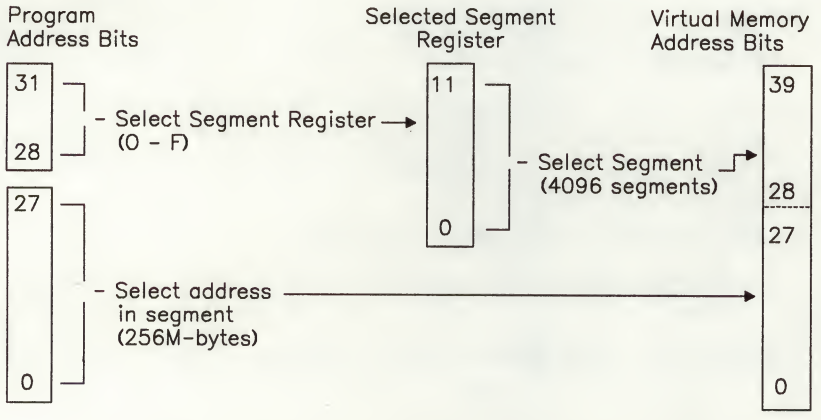


Figure 4-30. Segment Register Addressing

Segment Registers

The system uses a set of 16 segment registers for each process to select the areas of virtual memory that the process can use. Figure 4-31 shows how the system uses the segment registers. The VRM uses one of these registers (number 14) for its program space and another (register number 15) contains the addresses assigned to the input and output data bus hardware. Therefore, the rest of the operating system can use only registers 0 through 13. Of these registers, a process cannot address the kernel segment (register 0). Therefore, each process can access up to 13 different segments for a total available address space of approximately 3.5 billion bytes. The amount of actual memory and the paging space available on disk limit the addresses that can be used to a much smaller number that depends on system configuration.

Register	Addresses (Hex)	Used for
0	00000000 to 0FFFFFFF	Kernel code, data and stack Floating point code
1	10000000 to 1FFFFFFF	Current program code
2	20000000 to 2FFFFFFF	Current program static data and heap
3	30000000 to 3FFFFFFF	Current program stack
4 through 13	40000000 to DFFFFFFF	Shared segments accessed through shared memory system calls (**shm***xxx*) - 10 different shared segments per process **Note:** Segment register 13 maps the shared library. To avoid problems, do not use this register.
14	E0000000 to EFFFFFFF	VRM

Figure 4-31 (Part 1 of 2). Segment Register Usage

Register	Addresses (Hex)	Used for
15	F0000000 to FFFFFFFF	Hardware input/output addresses

Figure 4-31 (Part 2 of 2). Segment Register Usage

Using a Mapped Data File

Note: When using Distributed Services, a remote file cannot be used as a mapped data file.

The system provides a feature called ***mapped files*** to eliminate much of the overhead involved in writing and reading the contents of the files. This feature assigns the contents of the mapped file to an area of user memory. Once the program establishes this relationship, it can manipulate the file as if it were data in memory, using pointers to that data instead of input/output calls. However, the system does not detect end of file. If a program reads past the end of a file, it reads zeroes. The copy of the file on disk serves as the paging area for that file, saving paging space.

A program can use any regular file as a mapped data file. To create the mapped data file, perform the following actions:

1. Open (or create) the file and save the file descriptor:

```
if( ( fildes = open( filename , 2 ) ) < 0 )
{
        printf( "cannot open file\n" );
        exit(1);
}
```

2. Map the file to a segment using the **shmat** system call:

```
file_ptr = shmat( fildes, 0, SHM_MAP );
```

The parameter, **SHM_MAP**, is a constant defined in the header file, **/usr/include/sys/shm.h**. It indicates that the file is a mapped file. Include this header file and the other shared memory header files in a program with the directives:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

Note: The following steps show a method for detecting the end of a mapped file. The program must detect end of file in some manner.

3. Use the **lseek** system call to go to the end of file:

```
eof = file_ptr + lseek( fildes, 0, 2 );
```

This example sets the value of `eof` to an address that is 1 byte beyond the end of file. Use this value as the end of file marker in the program.

4. Now use `file_ptr` as a pointer to the start of the data file, and access the data as if it were in memory. The system calls for input and output, **read** and **write**, also work on the file, and produce the same data as when using pointers to access the data.

```
while( file_ptr < eof )
{
    .
    .
    .
    (references to file using file_ptr)
}
```

5. Close the file when the program is finished working with it:

```
close( fildes );
```

Using a Mapped Executable File

Note: When using Distributed Services, a remote file cannot be used as a mapped executable file. If it is a mapped executable file, the system treats it as a load style executable so that it can be run.

You can also extend the features of mapped data files to files containing compiled and executable object code. Because mapped files can be accessed more quickly than regular files, the system can load a program more quickly if its executable object file is a mapped file.

To create a program as a mapped executable file, compile and link the program using the **-K** flag with the **cc** or **ld** command. This flag tells the linker to create an object file with a page aligned format. That is, each part of the object file starts on a page boundary (an address that can be divided by 2K with no remainder). This option results in some empty space in the object file, but it allows the executable file to be mapped into memory. When the system maps an object file into memory, the text and data portions are handled differently:

Text This part of the program is mapped as a ***read only*** file, unless the program is running under the **sdb** program while debugging.

Data This part of the program is mapped as a ***copy on write*** file. Any changes made to the data are stored in the system paging area and are not written back to the original file (see "Copy on Write Mapped Files").

Copy on Write Mapped Files

To prevent the changes made to mapped files from appearing immediately in the file on disk, map the file as a ***copy on write*** mapped file. This option creates a mapped file with changes that are saved in the system paging space, instead of saving the changes to the copy of the file on disk. You must choose to write those changes to the copy on disk to save the changes. Otherwise, you lose the changes when closing the file.

Because the changes are not immediately reflected in the copy of the file that other users may access, use **copy on write** mapped files only among processes that cooperate with each other. See "Interprocess Communications" on page 4-29 for information about coordinating process activity.

If a program writes beyond the current end of file in a **copy-on-write** mapped file by storing into the corresponding memory segment (where the file is mapped), the actual file (on disk) is extended with blocks of zeroes in preparation for the new data. If the program does not perform an **fsync** system call before closing the file, the data written beyond the previous end of file is not written to disk. The file appears larger, but contains only the added zeroes. Therefore, always use an **fsync** system call before closing a **copy on write** mapped file to preserve any added or changed data.

To create a **copy on write** mapped data file, perform the following actions:

1. Open (or create) the file and save the file descriptor:

```
if( ( fildes = open( filename , 2 ) ) < 0 )
{
        printf( "cannot open file\n" );
        exit(1);
}
```

2. Map the file to a segment as ***copy on write***, using the **shmat** system call:

```
file_ptr = shmat( fildes, 0, SHM_CPY );
```

 The parameter, **SHM_CPY**, is a constant defined in the header file, **/usr/include/sys/shm.h**. It indicates that the file is a **copy on write** mapped file. Include this header file and the other shared memory header files in a program with the directives:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

3. Now use `file_ptr` as a pointer to the start of the data file, and access the data as if it were in memory.

4. Use the **fsync** system call to write changes to the copy of the file on disk to save the changes:

```
fsync( fildes );
```

5. Close the file when the program is finished working with it:

```
close( fildes );
```

Shared Memory Calls

Note: When using Distributed Services, remote memory cannot be shared.

The ***shared memory*** calls set aside an area of memory that cooperating processes can access. This area can serve as a large pool for exchanging data among the processes. The shared memory calls do not provide locks or access control among the processes. Therefore, processes using the shared memory area must set up a signal or semaphore control method to prevent access conflicts and to keep one process from changing data that another process is using. Use shared memory when the amount of data to be exchanged between processes is too large to transfer with messages, or when many processes maintain a common large database.

Use the following calls to create and use shared memory segments from a program:

Call	Description
shmctl	Controls shared memory operations.
shmget	Gets or creates a shared memory segment.
shmat	Attaches a shared memory segment to a process.
shmdt	Detaches a shared memory segment from a process.
disclaim	Removes mapping from a specified address range within a shared memory segment.

In addition, the **ftok** subroutine provides the ***key*** that the **shmget** call uses to create the shared segment. Include the following header files when using shared memory calls:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

Terms

The use of shared memory segments is similar in structure to the way the system creates and uses files. Defining the terms used for shared memory with respect to the more familiar file terms provides a framework to build an understanding of shared memory. Figure 4-32 shows the new terms and how they relate to terms used with files.

Term	Definition
key	The ***key*** is a unique identifier that names the particular shared segment. It is always associated with the shared segment as long as the shared segment exists. In this respect it is similar to the *filename* of a file.
shmid	The ***shmid*** is an identifier assigned to the shared segment for use within a particular process. It is similar in use to a *file descriptor* of a file.
attach	A process must ***attach*** a shared segment to use the shared segment. Attaching a shared segment is similar to opening a file.
detach	A process must ***detach*** a shared segment once it is finished with the shared segment. Detaching a shared segment is similar to closing a file.

Figure 4-32. Shared Memory Terms

Using Shared Segments

The following sequence describes the life cycle of a shared segment from initial creation to final removal from the system:

1. Create a key to uniquely identify the shared segment. Use the **ftok** subroutine to create the key. For example, to create a key `mykey` using a project ID of `R` contained in the variable `proj` (type **char**) and a file name of `null_file`, use a statement like:

   ```
   mykey = ftok( null_file, proj );
   ```

2. Either:

 - Create the shared memory segment with the **shmget** system call. For example, to create a shared segment that contains 4096 bytes and assign the **shmid** to an integer variable `mem_id`, use a statement like:

     ```
     mem_id = shmget( mykey, 4096, IPC_CREAT | 0666 );
     ```

 or

 - Get a previously created shared segment with the **shmget** system call. For example, to get a shared segment that is already associated with the key `mykey` and assign the **shmid** to an integer variable `mem_id`, use a statement like:

     ```
     mem_id = shmget( mykey, 4096, IPC_ACCESS );
     ```

3. Attach the shared segment to the process with the **shmat** system call. For example, to attach the previously created segment, use a statement like:

   ```
   ptr = shmat( mem_id );
   ```

 In this example, the variable `ptr` is a pointer to a structure that defines the fields in the shared segment. Use this template structure to store and retrieve data in the shared segment. This template should be the same for all processes using the segment.

4. Work with the data in the segment using the template structure.

5. Detach from the segment using the **shmdt** system call:

   ```
   shmdt( mem_id );
   ```

6. If the shared segment is no longer needed, remove it from the system using the **shmctl** system call:

```
shmctl( mem_id, IPC_RMID, ptr ) ;
```

See *AIX Operating System Technical Reference* for specific information about parameters for the calls and subroutines. You can also use the commands **ipcs** to get information about a segment and **ipcrm** to remove a segment. See *AIX Operating System Commands Reference* for information about these commands.

Memory Management Calls

Use the following calls to control a program's use of memory during execution:

Call	Description
brk	Change data segment space allocation
sbrk	Change data segment space allocation (see brk)

File System Calls

The system provides calls to create files, move data into and out of files, close files, and describe the restrictions and structure of the file system. Because the system treats input and output to all devices the same as input and output to files, you can use many of the ***file system*** calls for control of devices in the system also. The system calls do not provide the data formatting and housekeeping services that the C library subroutines for input and output do. See Chapter 3, "Using the Subroutine Libraries" on page 3-1 for information about the library calls.

If you use file system calls with Distributed Services, the translation tables must be set up to allow access to the remote files. See *IBM RT Managing the AIX Operating System* for information to set up the user ID, group ID and concurrent group ID translation tables.

Data Handling Calls

Use the following system calls to control data handling in the system. These calls create files, open and close files, and move data into and out of them.

Call	Description
close	Closes a file descriptor.
creat	Creates a new file or rewrite an existing one.
dup	Duplicates an open file descriptor.
ioctl	Controls device.
lseek	Moves read/write file pointer.
mknod	Makes a special file that provides an interface to a device for input and output.
open	Opens a file or device for reading or writing.
read	Reads from a file or device.
readx	Reads from a file or device and uses an extra parameter for communication with the device driver.
write	Writes on a file or device.
writex	Writes on a file or device and uses an extra parameter for communication with the device driver.

Using Files

When transferring data on the system, ensure that the files are created, opened and closed at the proper times so that data is not lost. The following sequence of events describes the actions to perform when using files for data storage:

1. Use the **creat** system call to create the file

 or

 Use the **open** system call to open the file if it already exists (the **creat** call leaves the file open).

2. Use the **write** and **read** system calls to transfer data into and out of the file.

3. Use the **close** system call to close the file.

In most cases, closing the file is enough. The system writes the file to disk from its buffers in memory at its own convenience. However, to write the data to disk immediately, use the **fsync** system call to force the system to write its buffers to disk before closing the file.

File Descriptors

The system performs all input and output by reading or writing files. It uses ***special files*** to form the interface between a device and the operating system. When you open a file, the system checks to see if you can access it. If you have access to the file, the system returns a small positive integer called a ***file descriptor***. The system uses this file descriptor instead of the name to identify the file. Therefore, programs must use the file descriptor when doing input and output with system calls.

The system assigns the file descriptor number on an *as available* basis, but it reserves three numbers for special functions:

0 ***standard input***: This file normally handles input from the keyboard of the terminal.

1 ***standard output***: This file normally handles output to the terminal screen.

2 ***standard error***: This file normally handles error messages to the terminal screen.

These file descriptors are normally always open, so that a program can get input from the standard input and send output to standard output or error without opening a file.

Opening and Closing Files

To use input and output other than **stdin**, **stdout**, or **stderr**, either **open** or **creat** a file. The **open** system call sets up an existing file for access, and returns a file descriptor to the calling program:

```
int fildes;
fildes = open (filename,oflag,[mode]);
```

In this call, the parameters have the following meaning:

filename The character string that corresponds to the external file name of the file to be opened.

oflag A flag that indicates the conditions of the access for the file.

mode The optional access permission bits for the file. These bits are used only if the file must first be created before it can be opened. See the **chmod** command in *AIX Operating System Commands Reference* for an explanation of the access permission bits.

If an error occurs, **open** returns -1 as the file descriptor. Trying to open a file that does not exist is an error.

To create a new file, use the **creat** system call:

```
fildes = creat(filename,mode);
```

In this call, the parameters have the following meaning:

filename The character string that corresponds to the external file name of the file to be opened.

mode The access permission bits for the file to be created as described for the **chmod** command in *AIX Operating System Commands Reference.*

The **creat** call returns -1 as the file descriptor if it cannot create the file. If the file exists, this call truncates that file to zero length and returns the file descriptor for that file.

To free up a file descriptor for use with another file, use the **close** system call when access to the file is complete:

```
close(fildes);
```

When the program stops using an **exit** call, or a **return** from **main**, the system closes all file descriptors associated with the program.

Random Access to Files

Normal access to files is sequential. Each **read** or **write** occurs in the file position directly following the previous operation. To perform random access I/O, use the **lseek** system call to move around in the file. This system call does not read or write to the file; it only changes the position where the next **read** or **write** will occur. The format of this call is:

```
lseek(fildes, offset, whence);
```

In this call, the parameters have the following meaning:

fildes The file descriptor for the file.

offset The number of bytes to move, or an absolute address as specified by the *whence* parameter.

whence Determines how to use *offset* to move in the file:

0	Moves to the address contained in *offset*
1	Moves to the address that is the current location plus the value contained in *offset.*
2	Moves to the address that is the number of bytes contained in *offset* plus the address of the end of the file.

For example, to append to a file when it is not positioned at the end, seek to the end before writing:

```
lseek(fildes, 0, 2);
```

To get back to the beginning of the file:

```
lseek(fildes, 0, 0);
```

To create ***sparse files***, or files with ***holes*** in them to allow for relative record access within the file, create a new file and then use the **lseek** call to move to selected places in the file before writing. The spaces between the data (***holes***) become part of the file, but do not take up disk space until they are actually filled with data.

Reading and Writing to a File

The **read** and the **write** system calls perform input and output for the system. These calls require three arguments:

- **File descriptor**: The integer assigned to the file involved in the read or write.
- **Buffer**: An area in the program that supplies or receives the data.
- **Byte count**: The number of bytes to be transferred.

Each call returns the number of bytes actually transferred. For a **read**, this number may be less than the number requested in the byte count parameter. For a **write**, if this number is not the number requested, an error occurred. A returned value of 0 indicates the end of file; a returned value of -1 indicates an error occurred during the operation.

The number of bytes to transfer is the programmer's choice. Some useful values are:

1 One character at a time, or *unbuffered* transfer.

512 The block size for many peripheral devices.

1024 The internal block size for the operating system. This size, or a multiple of this size, is efficient for normal operations.

Using the Extended Calls

The system provides extended versions of the following system calls:

readx

writex

These calls perform the same functions as their original versions, but they also provide an extra parameter to pass information to the device driver. How the parameter is used depends on the device driver with which the calls are used. The parameter can be used either as a value or a pointer to a buffer area containing additional information. Use them only with device drivers that understand the additional information.

File Maintenance Calls

Use the file maintenance calls for programs that change protection of files in the file system, access many different files, or provide control of files for the user of the program. Many of these calls are the base for the system commands that have similar names. You can, however, use these calls to write new commands or utilities to help in the program development process, or to include in an application program. The file maintenance calls on the system include:

Call	Description
access	Determines accessibility of a file.
chdir	Changes working directory.
chmod	Changes mode of a file.
chown	Changes owner and group of a file.
chroot	Changes root directory.
fclear	Clears space in a file.
fcntl	Controls file operations.
fstat	Gets data returned by **stat** system call.
fsync	Forces changes in a file to disk.
ftruncate	Makes a file shorter.
link	Links to a file.
lockf	Locks a region of a file, or provides exclusive regions in a file.
mount	Mounts a file system.
stat	Gets file status.
sync	Updates superblock.
umask	Sets and gets file creation mask.
umount	Unmounts a file system.
unlink	Unlinks a file.
ustat	Gets File system statistics.
utime	Sets file access and modification times.

Time System Calls

The system provides the following calls to set the system time and to find out what the system time is. See the description for the **ctime** library routine in *AIX Operating System Commands Reference* to get formatted time data from the system.

Call	Description
stime	Sets time.
time	Gets time.

Both calls use a value of time that is the number of seconds since 00:00:00 Greenwich Mean Time (GMT) on January 1, 1970. Therefore, the program must be able to calculate the date using that starting date and the elapsed time value used by the time calls, total seconds. For convenience, some of the common time units converted to seconds are:

Unit	Value in Seconds
minute	60
hour	3600
day	86,400
week	604,800
month	2,419,200 (28 days)
	2,505,600 (29 days)
	2,592,000 (30 days)
	2,678,400 (31 days)
year	31,536,000 (365 days)
	31,622,400 (366 days)

Using System Calls with Distributed Services

When using Distributed Services, you can share files and resources of RT work stations on a network as if they were local files and resources. The Distributed Services commands and subroutines distribute kernel functions across physical system boundaries. They retain AIX Operating System file system semantics while distributing data access across a network. The tree-structured file hierarchy can be composed of files and directories that can reside locally or across many different network RT nodes. The actual location of the file components is not evident to the user or application program.

To control access to information across the distributed file system, Distributed Services uses the access permission system of AIX Operating System (user IDs, group IDs and passwords) together with the network access and BIND passwords used by SNA Services. It also allows the administrator to define and update user network information contained in a user profile. (For more information on user profiles, refer to *IBM RT Using the AIX Operating System.*)

Distributed Services also allows programs to use interprocess communication (IPC) message queues (see "Interprocess Communications" on page 4-29). Application programs can use these queues to exchange information or coordinate activity between programs running on separate systems. Application programs that take advantage of this facility can access local and remote queues as if they were local queues. End users of these application programs are not aware of the actual location of the queue(s).

You can use the existing system calls for accessing files and message queues with Distributed Services. Therefore, existing application programs that access files and IPC message queues using AIX Operating System system calls need not be changed. Some system calls have been enhanced with new capabilities. In addition, some new system calls have been added to further support a distributed network environment.

The mount capability allows transparent access of remote files. For example, a local program can mount a remote directory (located on a remote system) over a local directory. Files below the mounted remote directory can then be accessed by a path that may start at the local system and cross the *remote directory over local directory* mount point. The **mount** command allows a remote directory (or file) to be mounted on the local file system. A system call, **vmount**, is used by the **mount** command to mount directories or files from a remote system. Another system call, **fullstat**, gives additional information about remote files that the **stat** system call does not.

When using Distributed Services, a program can use distributed files and IPC message queues. It cannot distribute all interprocess communication methods, such as semaphores, shared memory, or mapped data files. In addition, it cannot fork a process into a remote node.

Distributed Services Example Program

The following example requires that Distributed Services is installed, configured and running properly, and that the remote system is a node that is defined to the local system. Refer to *IBM RT Managing the AIX Operating System* for information to correctly configure Distributed Services.

In addition, to run the example program you must have the permissions required on your system to create a directory in **/tmp** and to mount a file system. In addition, you must have the permissions required to create a file in the **/tmp** directory on the remote system.

The following example program shows how to use Distributed Services system calls to access files in a Distributed Services network. It uses the **mount** command to mount a remote directory and the **fullstat** system call to get information about a remote file. Except for the use of these two calls, the program uses the same system calls for input and output to the remote system as would a program performing input and output on the local system. If you are using the Programming Examples, the program is stored as **distserv.c**. Compile the program to an executable file named **distserv** with the following command:

```
cc distserv.c -o distserv
```

You can then run the program by entering the command:

```
distserv nodeid
```

In this format *nodeid* is the numerical node ID of the remote system. You can get this node ID by using the **uname -a** command on the remote system. For example, if the ID of the remote node is 108133EB, enter the command:

```
distserv 108133EB
```

When the program runs, it:

1. Makes a new directory in the local **/tmp** directory.
2. Uses the **mount** command to mount a remote directory over the new local directory.
3. Uses the **open** system call to create a file, **dsfile**, in the remote directory.
4. Uses the **write** system call to write 15 records to the file.
5. Uses the **close** system call to close the file.
6. Uses the **fullstat** system call on the new file to get information about the file.
7. Prints the location, name and size of the remote file.
8. Uses the **umount** command to unmount the remote directory.

Figure 4-33 on page 4-88 shows the program listing for **distserv.c**. Figure 4-34 on page 4-92 shows the output produced when the program runs. When the program completes successfully, it creates a test file, called **dsfile**, in the **/tmp** directory of the

remote system. That file contains a string of digits (60 copies of the string 1234567890). You can look at that file by mounting the remote directory again (the program unmounted it when it finished). For the preceeding example, the mount command is:

```
mount -n 108133EB /tmp /tmp/remote
```

To look at the file, use the following command:

```
pg /tmp/remote/dsfile
```

When you are done with the program, erase the remote file, unmount the remote file system, and delete the mounting directory:

```
rm /tmp/remote/dsfile
umount /tmp/remote
rmdir /tmp/remote
```

```
#include <sys/types.h>
#include <sys/fullstat.h>
#include <sys/vmount.h>
#include <sys/errno.h>
#include <fcntl.h>

#define  RECORDSIZE     40

   char    valid_chars[] = "0123456789abcdefABCDEF";
   char cmdbuf [256]
   static  char record[] = "12345678901234567890123456789012345678901234567890";

   main(argc, argv)
   int argc;
   char **argv;
   {
      int fildes;              /* file descriptor */
      int i, rc;               /* for loop index and return code */
      struct fullstat fsbuf;   /* fullstat buffer */
      int errflag = 0;
      extern errno;
      unsigned long nid = 0;   /* the integer form of the nid       */
      unsigned long *info = NULL;
                               /* pointer to data area for vmount() */
      extern char *malloc();
      extern int vmount();
```

Figure 4-33 (Part 1 of 5). Example Program Listing for distserv.c

```
if( argc != 2 )
{
    usage();
    exit();
}
printf("Making directory \"remote\" in local /tmp directory.\n");
if( mkdir("/tmp/remote", 777) )
{
    perror("mkdir(2) failed ");
    if( errno != EEXIST )
        exit(1);
}
printf("Checking the specified node id.\n");
rc = chk_nname(argv[1],&nid);
if( rc == -1 )
{
    printf("Nodename is not valid\n");
    exit(1);
}
if( ( info = (unsigned long *)
     malloc(sizeof(unsigned long))) == NULL )
{
    perror("malloc(3) failed ");
    exit(1);
}
else
   *info = nid;
```

Figure 4-33 (Part 2 of 5). Example Program Listing for distserv.c

```
sprintf(cmdbuf,"%s%x %s%s",""mount -n", info, "
    /tmp","/tmp/remote")
if ((system (cmdbuf))==-1)
{
    perror("vmount(2) failed ");
    exit(1);
}
printf("Creating file \"dsfile\" in the remote directory.\n");
if( (fildes=open("/tmp/remote/dsfile", O_RDWR|O_CREAT|O_TRUNC, 666))
    == -1 )
{
    perror("open(2) failed ");
}
else
{
    printf("Writing 15 records to the remote file.\n");
    for( i=1; i<=15; i++ )
    {
        if( (rc = write(fildes, record, RECORDSIZE)) == -1 )
        {
            perror("write(2) failed ");
            errflag++;
        }
    }
    printf("Closing the remote file /tmp/remote/dsfile.\n");
    close(fildes);
```

Figure 4-33 (Part 3 of 5). Example Program Listing for distserv.c

```
        if( !errflag )
        {           /* do a fullstat on the file */
            printf("Getting information about the remote file.\n");
            if( fullstat("/tmp/remote/dsfile", FL_STAT, &fsbuf) < 0 )
            {
                perror("fullstat(2) failed ");
            }
            else
            {
                printf("Name of the file: /tmp/remote/dsfile\n");
                printf("Node of the file: %x\n", fsbuf.fst_nid);
                printf("Size of the file: %d bytes\n", fsbuf.st_size);
            }
        }
    printf("Unmounting the remote directory /tmp/remote.\n");
    sprintf(cmdbuf,"%s","umount /tmp/remote);
    if system (cmdbuf);
    }
}

usage()
{
    printf("usage: distserv <node id>\n");
}
```

Figure 4-33 (Part 4 of 5). Example Program Listing for distserv.c

```
int chk_nname(nodename,nid)
char *nodename;
unsigned long *nid;
{
    register char *testp;   /* used in nodename validation          */
    int length;             /* length of nodename                   */
    int i;                  /* counter and error returns            */

    testp = (char *) ~NULL;
    length = strlen(nodename);
    for( i=0; (i<length) && (testp); i++ )
            testp = (char *)strchr( valid_chars, nodename[i] );
    if( (testp == 0) || (strlen(nodename) != 8) )
            return(-1);
    sscanf( nodename, "%x", nid);
    return(0);
}
```

Figure 4-33 (Part 5 of 5). Example Program Listing for distserv.c

```
Making directory "remote" in local /tmp directory.
Checking the specified node id.
Mounting remote /tmp directory on local /tmp/remote.
Creating file "dsfile" in the remote directory.
Writing 15 records to the remote file.
Closing the remote file /tmp/remote/dsfile.
Getting information about the remote file.
Name of the file: /tmp/remote/dsfile
Node of the file: 108133eb
Size of the file: 600 bytes
Unmounting the remote directory /tmp/remote.
```

Figure 4-34. Output from Example Program distserv.c

Using File Locking

AIX Operating System allows many processes to synchronize simultaneous access to a file through the use of the **fcntl** and **lockf** system calls. These system calls allow a program to lock and unlock portions of an open file. The program can use either a read lock or a write lock.

A ***read lock*** prevents any other process from setting a write lock on any portion of the protected area. When a read lock is set on a segment of a file, other processes can also set read locks on that segment or a portion of it. The file descriptor on which a read lock is being placed must have been opened with read access. Use the **fcntl** system call to set a read lock on a file.

A ***write lock*** prevents any other process from setting a read lock or a write lock on any portion of the protected area. Only one write lock and no read locks can exist for a given segment of a file at a given time. The file descriptor on which a write lock is being placed must have been opened with write access. Use the **fcntl** system call or the **lockf** system call to set write locks.

All locks associated with a file for a given process are removed when the file descriptor for that file is closed by that process or when the process holding the file descriptor ends. Locks are not passed to a child process after executing a **fork** system call.

File locks can operate as either ***enforced*** or ***advisory*** locks. Enforced locks prevent other processes from accessing the locked area of the file. Advisory locks are not enforced by the operating system and require cooperating processes that check the lock bits of files to ensure data integrity. To select enforced locking, the S_ENFMT code must be set in the access permission bits (or mode) of the file. Using the **chmod** system call on a file that has locks can change the type of lock between forced and advisory. Otherwise, locking is advisory. Thus, a particular file can have advisory or enforced locks, but not both. Use advisory locks instead of enforced locks whenever possible.

A process cannot use a **write** system call on any region of a file that is protected by an enforced read lock or an enforced write lock that is held by another process. A process cannot use a **read** system call on any region of a file that is protected by an enforced write lock that is held by another process. In addition, the **creat**, **open**, **ftruncate** and **fclear** system calls cannot truncate a file protected by any enforced lock that is held by another process.

When an area of a file is protected by an enforced lock and another process attempts to access that area of the file, the attempt is not successful but the result varies with other process conditions. A **read** or **write** system call that is blocked may either sleep until the area is unlocked or, if the NO_DELAY flag for that file descriptor is set, return with an error. However, if the system detects that sleeping would cause deadlock, then the system call fails with **errno** set to EDEADLK. If another process attempts to truncate the file with either the **creat** or **open** system call, that system call fails with **errno** set to EACCES.

The **read**, **write**, **creat** and **open** system calls are not affected by advisory locking. Processes must voluntarily call **lockf** to make advisory locks effective.

File and record locking helps preserve data integrity when multiple processes are using the same file. When using Distributed Services, this protection applies to both local and remote files. Processes that use record and file locking in a distributed (multiple node) environment must be careful not to introduce permanent deadlocks when they wait for locks. Before waiting for a lock, these processes should set a timer so that they will be awakened by a signal if there is the possibility of a deadlock.

File Locking Example Program

The following example program shows how to use locking system calls. If you are using the Programming Examples, the program is stored as **locktest.c**. Compile the program to an executable file named **locktest** with the following command:

```
cc locktest.c -o locktest
```

You can then run the program by entering the command:

```
locktest filename
```

In this format, *filename* is the name of a temporary file that the program creates. Ensure that this name is not the name of any file in the current directory. For example, to start the program and allow it to create a file named **testfile**, enter the command:

```
locktest testfile
```

When the program runs, it:

1. Creates a signal handler to catch SIGALRM if it occurs.
2. Creates the test file with enforced locking enabled using the **open** system call.
3. Uses the **lockf** system call to enforce lock the whole file from the parent process.
4. Uses the **fork** system call to create a child process.
5. Writes 15 records to the file from the parent process while the child process tries to lock one record in the file.
6. The child process sleeps until the parent process unlocks the file.
7. The parent process unlocks the file.
8. The child process wakes up and completes its record lock.
9. The child process reads one record from the file.
10. The child process removes its record lock from the file and then exits.
11. The parent process detects the end of the child process and exits.

Figure 4-35 shows the program listing for **locktest.c**. Figure 4-36 on page 4-99 shows the output produced when the program runs. When the program completes successfully, it creates a test file that contains a string of digits. When you are done with the program, erase the test file.

```
#include <fcntl.h>
#include <sys/lockf.h>
#include <sys/stat.h>
#include <sys/signal.h>
#include <errno.h>

#define  TIMEOUT        10
#define  RECORDSIZE     40
#define  RECORD(x)      (x-1)*RECORDSIZE

static char record[] = "This string has 40 characters w/newline\n";

salrm()         /* signal handler for receiving SIGALRM */
{
   printf("SIGALRM signal received.\n");
}

main(argc, argv)
int argc;
char **argv;
{
   int fildes;              /* file descriptor */
   int i, rc;               /* for loop index and return code */
   char buf[RECORDSIZE];    /* read buffer */
   static struct sigvec alrmvec = {salrm, 0, 0};
```

Figure 4-35 (Part 1 of 4). Example Program Listing for locktest.c

```
if( argc != 2 )
{
   usage();
   exit();
}
if( sigvec(SIGALRM, &alrmvec, NULL) == -1 )
{
   perror("sigvec(2) failed ");
   exit(1);
}
if( (fildes=open(argv[1], O_RDWR | O_CREAT | O_TRUNC,
    S_ENFMT | S_IRUSR | S_IWUSR ))== -1 )
{
   perror("open(2) failed ");
   exit(1);
}
alarm(TIMEOUT);
printf("Parent process tries to lock the entire file.\n" );
if( lockf(fildes, F_LOCK, 0) == -1 )
{
   perror("parent process lockf(2) failed ");
   exit(1);
}
alarm(0);
printf("Fork the child process.\n" );
switch(fork())
{
case -1:                              /* fork error */
   perror("fork(2) failed ");
   exit(1);
```

Figure 4-35 (Part 2 of 4). Example Program Listing for locktest.c

```
case 0:                                  /* child process */
   printf("Child process tries to lock the 5th record in the file.\n" );
   lseek(fildes, RECORD(5), 0);
   alarm(TIMEOUT);
   if( lockf(fildes, F_LOCK, RECORDSIZE) == -1 )
   {
      perror("child process lockf(2) failed ");
      exit();
   }
   alarm(0);
   if( (rc = read(fildes, &buf[0], RECORDSIZE)) == -1 )
   {
      perror("child process read(2) failed ");
      exit();
   }
   printf("Child process read record 5 from the file.\n" );
   printf("Child process tries to unlock the 5th record in the file.\n" );
   lseek(fildes, RECORD(5), 0);
   alarm(TIMEOUT);
   if( lockf(fildes, F_ULOCK, RECORDSIZE) == -1 )
   {
      perror("child process lockf(2) failed ");
      exit();
   }
   alarm(0);
   close(fildes);
   exit(0);
```

Figure 4-35 (Part 3 of 4). Example Program Listing for locktest.c

```
   default:                                /* parent process */
      for( i=1; i<=15; i++ )
      {
         if( (rc = write(fildes, record, RECORDSIZE)) == -1 )
         {
             perror("parent process write(2) failed ");
             exit();
         }
         printf("Parent process wrote record %d to the file.\n" , i);
      }
      printf("Parent process tries to unlock the entire file.\n" );
      lseek(fildes,0,0);
      alarm(TIMEOUT);
      if( lockf(fildes, F_ULOCK, 0) == -1 )
      {
         perror("parent process lockf(2) failed ");
         exit(1);
      }
      alarm(0);
      printf("Parent process waits for the child to end.\n" );
      wait(0);
      printf("Child process has ended; Parent process ends.\n" );
      close(fildes);
      exit(0);
   }
}

usage()
{
   printf("usage: locktest <file>\n");
}
```

Figure 4-35 (Part 4 of 4). Example Program Listing for locktest.c

```
Parent process tries to lock the entire file.
Fork the child process.
Child process tries to lock the 5th record in the file.
Child process read record 5 from the file.
Child process tries to unlock the 5th record in the file.
Parent process tries to lock the entire file.
Fork the child process.
Parent process wrote record 1 to the file.
Parent process wrote record 2 to the file.
Parent process wrote record 3 to the file.
Parent process wrote record 4 to the file.
Parent process wrote record 5 to the file.
Parent process wrote record 6 to the file.
Parent process wrote record 7 to the file.
Parent process wrote record 8 to the file.
Parent process wrote record 9 to the file.
Parent process wrote record 10 to the file.
Parent process wrote record 11 to the file.
Parent process wrote record 12 to the file.
Parent process wrote record 13 to the file.
Parent process wrote record 14 to the file.
Parent process wrote record 15 to the file.
Parent process tries to unlock the entire file.
Parent process waits for the child to end.
Child process has ended; Parent process ends.
```

Figure 4-36. Output from Example Program locktest.c

Distributed Services Message Queue Example

The following example programs show the use of message queues operating with Distributed Services. The example consists of three programs. If you are using the Programming Examples, the programs are stored as:

msgqcrt.c A program to create an IPC queue for the programs to use. Figure 4-37 on page 4-101 shows the program listing for **msgqcrt.c**.

msgqsnd.c A program that sends a series of messages to the defined queue. Figure 4-38 on page 4-103 shows the program listing for **msgqsnd.c**.

msgqrcv.c A program that receives the series of messages from the defined queue. Figure 4-39 on page 4-105 shows the program listing for **msgqrcv.c**.

The programs are adapted from the example program shown in "Example of Message Queue Calls" on page 4-62. Refer to that section for a description of the general operation of the programs.

Compile the programs to executable files with the following commands:

```
cc msgqcrt.c -o msgqcrt -lipc
cc msgqsnd.c -o msgqsnd -lipc
cc msgqrcv.c -o msgqrcv -lipc
```

To run the programs on a single machine, see "Running the Programs in a Local Environment" on page 4-107. To run the programs on two separate machines using Distributed Services, first read the instructions for running on a single machine and then see "Running the Programs in a Distributed Services Environment" on page 4-110.

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <drs.h>
#define MSGSIZ          15      /* common size                      */
#define MSGRTYP          0      /* receive first message on queue */
#define MTYPE            1      /* common message type              */

int  msgqid;
int error_num;
```

Figure 4-37 (Part 1 of 2). Message Queue Creation Example Program

```
main ()
{
  key_t key, rkey;
  char nid[15];
  char *receive();
  int msgqid;

  if( find_ipc_prof( "msgtestQ", &key, &rkey, nid ) < 0)
  {
      if( (error_num = create_ipc_prof( "msgtestQ", &key, &rkey, nid )) < 0)
      {
           printf( "Could not create ipc profile; Error=%d\n", error_num );
           exit(0);
      }
      printf( "Created a message queue with queue name = msgtestQ.\n" );
  }
  else
  {
      printf( "A message queue with queue name = msgtestQ already exists.\n" );
  }
  msgqid = msgget( key, IPC_CREAT | S_IRWXU | S_IRWXG | S_IRWXO );
  printf( "Local key for queue is: %d\n", key );
  printf( "Remote key for queue is: %d\n", rkey );
  exit(0);
}
```

Figure 4-37 (Part 2 of 2). Message Queue Creation Example Program

```
#include <stdio.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <drs.h>
#define MSGSIZ          15      /* common size                      */
#define MSGRTYP          0      /* receive first message on queue */
#define MTYPE            1      /* common message type              */

static char *msg[11] = {
                         "this is item 0", "this is item 1",
                         "this is item 2", "this is item 3",
                         "this is item 4", "this is item 5",
                         "this is item 6", "this is item 7",
                         "this is item 8", "this is item 9",
                         "this is item 10"
                       };
int  msgqid;
extern int errno;

main()
{
  key_t key, rkey;
  char nid[15];
  char *receive();
  int send();
  int i ;
  int  rtn;
```

Figure 4-38 (Part 1 of 2). Send Message Example Program

```
  if( (rtn = find_ipc_prof("msgtestQ",&key,&rkey,nid )) != 0)
  {
       printf( "Could not find an ipc profile for msgtestQ: %d\n", rtn );
       printf( "Please run program msgqcrt to create the queue.\n" );
       exit(0);
  }
  if( (msgqid = msgget( key, S_IRWXU | S_IRWXG | S_IRWXO )) < 0)
  {
       perror( "msgqsnd cannot open message queue" );
       exit(0);
  }
  printf( "msgqid = %d\n", msgqid );
  for( i = 0;i <= 10;i++ )
  {
       printf( "Producer process: sending message number %d\n", i );
       send(msg[i]);       /* produce item */
  }
  printf( "Producer process ended\n");
  exit(0);
}   /* End Producer Process */

struct msgbuf msgb, *msgp = &msgb;   /* pointer to message buffer */
char *strcpy();

send( item )
char *item;
{
  msgp->mtype = MTYPE;
  strcpy(( msgp->mtext ), item );
  msgsnd( msgqid, msgp, MSGSIZ, ~IPC_NOWAIT );
  return;
}
```

Figure 4-38 (Part 2 of 2). Send Message Example Program

```
#include <errno.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <drs.h>
#define MSGSIZ          15      /* common size                      */
#define MSGRTYP          0      /* receive first message on queue */
#define MTYPE            1      /* common message type              */

int  msgqid;
extern int errno;

main ()
{
  key_t key, rkey;
  char nid[15];
  char *receive();
  int i;
  int rtn;

  if ( (rtn = find_ipc_prof( "msgtestQ", &key, &rkey, nid )) < 0)
  {
       printf( "Could not find an ipc profile for msgtestQ: %d\n", rtn );
       printf( "Please run program msgqcrt to create the queue.\n" );
       exit(0);
  }
```

Figure 4-39 (Part 1 of 2). Receive Message Example Program

```
  if( (msgqid = msgget( key, IPC_CREAT | S_IRWXU | S_IRWXG | S_IRWXO )) < 0)
  {
       perror( "msgqrcv cannot open message queue" );
       exit(0);
  }
  printf( "msgqid = %d\n", msgqid );
  for( i = 0;i <= 10;i++ )
  {
      printf( "Consumer process: message number %d is: %s\n",
                  i, receive() ); /* consume item */
  }
  printf( "Consumer process ended\n");
  (void) msgctl (msgqid, IPC_RMID, NULL);
  exit(0);
}

struct msgbuf msgb, *msgp = &msgb;   /* define pointer to message buffer */
char *receive()
{
  msgrcv( msgqid, msgp, MSGSIZ, MSGRTYP, ~IPC_NOWAIT );
  return(( msgp->mtext ));
}
```

Figure 4-39 (Part 2 of 2). Receive Message Example Program

Running the Programs in a Local Environment

After compiling the programs, you can run them at a single machine to see the effects of the message queue calls. To run them from a single terminal, enter the commands in the following order:

```
msgqcrt
msgqsnd
msgqrcv
```

You must run **msgqcrt** to create the message queue before running either of the other programs. If you do not create the queue first, the other programs quit with the following error message when you run them:

```
msgqsnd cannot open message queue: No such file or directory
```

You can also run **msgqrcv** before running **msgqsnd** but you must run them from separate virtual (or physical) terminals. Use the following sequence of commands and keys to run the programs from two virtual terminals (press the **Alt** and **Action** keys to switch between virtual terminals):

1. Open a second virtual terminal:

   ```
   open sh
   069-031 You have opened the "sh" window.
   ```

2. Run **msgqcrt** from the second virtual terminal:

   ```
   ./msgqcrt
   ```

 The output from this program is shown in Figure 4-40 on page 4-108 or Figure 4-41 on page 4-108, depending upon whether or not the queue exists already.

3. Run **msgqrcv** from the second virtual terminal:

   ```
   ./msgqrcv
   ```

 The command displays the first part of the output shown in Figure 4-43 on page 4-109 and waits for data in the queue.

4. Run **msgqsnd** from the first virtual terminal:

   ```
   ./msgqsnd
   ```

 The command displays the output shown in Figure 4-42 on page 4-108 on the first terminal, and completes the output for **msgqrcv** on the second terminal.

```
Created a message queue with queue name = msgtestQ.
Local key for queue is: 196608
Remote key for queue is: 0
```

Figure 4-40. Message Queue Creation Example Program Output

```
A message queue with queue name = msgtestQ already exists.
Local key for queue is: 196608
Remote key for queue is: 0
```

Figure 4-41. Message Queue Already Exists Example Program Output

```
msgqid = 327687
Producer process: sending message number 0
Producer process: sending message number 1
Producer process: sending message number 2
Producer process: sending message number 3
Producer process: sending message number 4
Producer process: sending message number 5
Producer process: sending message number 6
Producer process: sending message number 7
Producer process: sending message number 8
Producer process: sending message number 9
Producer process: sending message number 10
Producer process ended
```

Figure 4-42. Send Message Example Program Output

```
msgqid = 327687
Consumer process: message number 0 is: this is item 0
Consumer process: message number 1 is: this is item 1
Consumer process: message number 2 is: this is item 2
Consumer process: message number 3 is: this is item 3
Consumer process: message number 4 is: this is item 4
Consumer process: message number 5 is: this is item 5
Consumer process: message number 6 is: this is item 6
Consumer process: message number 7 is: this is item 7
Consumer process: message number 8 is: this is item 8
Consumer process: message number 9 is: this is item 9
Consumer process: message number 10 is: this is item 10
Consumer process ended
```

Figure 4-43. Receive Message Example Program Output

Running the Programs in a Distributed Services Environment

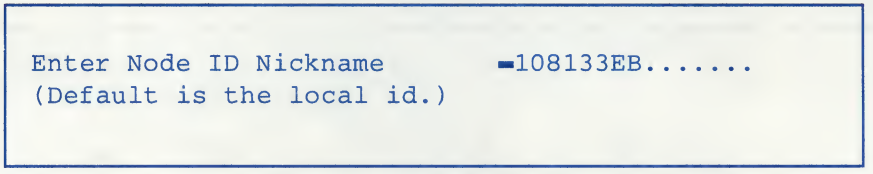
To run the example programs to pass messages across a network using Distributed Services, you must configure the IPC queue tables at each system to recognize the distributed queue. The following procedure provides steps to configure the IPC queue tables and to run the programs in a distributed environment. Refer to *IBM RT Managing the AIX Operating System* for more information about configuring distributed message queues.

1. Create the message queue on one of the two systems (this system is called the ***local system*** in the remainder of this discussion). Write down the local key value that the program displays on the screen.

   ```
   $ ./msgqcrt
   ```

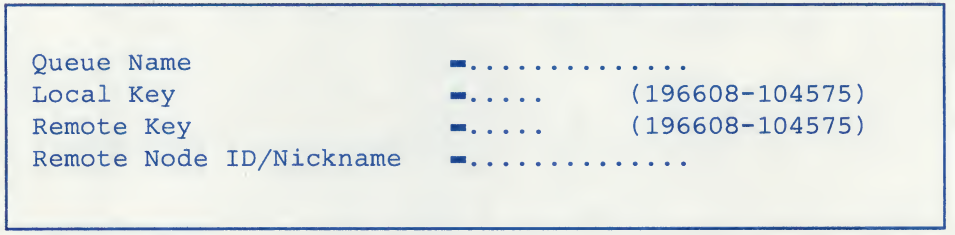
2. Use the **ipctable** command on the remote system (the system that does not have the queue installed) to define the queue to the remote system as follows:

 a. Enter the **ipctable** command. A pop-up panel appears that displays the node ID. For the system used in this example, the panel appears as follows:

 b. Press **Do**. The Distributed IPC Queues Table appears.

 c. On the command bar, select ADD. A pop-up panel appears that requests information about the queue that you want to add:

Type in the following information in these fields:

Field	Value
Queue Name	msgtestQ
Local Key	Leave this field blank. The system assigns a local key value.
Remote Key	Enter the local key value returned from the **msgqcrt** program on the local system.
Remote ID/Nickname	Enter the node ID value for the local system (the system where the queue was defined).

For the example programs running on the two systems that created the output examples, the pop-up panel looks like:

```
Queue Name                 =msgtestQ......
Local Key                  =......      (196608-104575)
Remote Key                 =196608      (196608-104575)
Remote Node ID/Nickname    =208131AA......
```

d. Press **Do**. The system assigns a local key to the queue and adds the queue to the Distributed IPC Queues Table.

e. Select CLOSE from the command bar to return to the AIX Operating System command line.

3. Run the **msgqsnd** program on the remote system and the **msgqrcv** program on the local system. The same output appears for each command as when they were run on one system. You could also run **msgqrcv** on the remote system and **msgqsnd** on the local system. However, be sure to run **msgqcrt** on one system before each time you run the other two programs.

Chapter 5. Controlling the Terminal Screen

CONTENTS

About This Chapter 5-2
Introduction 5-3
- New Terms 5-4
- What You Need 5-5
- Using the Screen Update Routines 5-6
- What the Screen Looks Like 5-7
- Function Names 5-9
- Variables 5-10
- Extended Characters 5-11

Using the Library Routines 5-12
- Setting Up the Environment 5-12
- Writing to a Window 5-13
- Getting Input from the Terminal 5-16
- Controlling the Screen 5-17

Routines for Panels and Panes 5-21
- Defining Panels and Panes 5-21
- Creating Panels and Panes 5-23

Display Attributes 5-27
- Changing the Defined Attributes 5-29
- Changing Screen Attributes 5-31

Using Other Features 5-33
- Controlling Input with the keypad, extended, and trackloc Routines 5-33
- Scrolling Windows 5-35
- Improving Performance 5-36

Example Program 5-36

About This Chapter

The system contains two libraries of routines to support input and output to the terminal screen. These libraries are:

curses A set of screen control routines. This library is included for compatibility with existing application programs.

extended curses An enhancement to the **curses** set of routines for IBM RT that provides extended function for:

- Expanded character set
- Color
- Multiple character attributes
- Error detection and handling
- Efficient handling of a window-oriented screen presentation, including:
 - Window stacking and layers
 - Linked scrolling of windows
 - Scrolling data in windows that are partially covered
 - Automatic tracking of active panes.
- 2-byte characters for international character support
- **locator** input support

Use these routines for new program development or to increase the function of existing programs.

This chapter discusses only the extended curses library. Information about both libraries is in *AIX Operating System Technical Reference*.

Introduction

The extended curses library contains a set of C language routines that:

- Updates a screen.
- Gets input from the terminal in a screen-oriented fashion.
- Moves the cursor from one point to another independent of other screen activities.
- Creates and manages a screen containing windows, panels and panes.

The routines do the most common type of terminal-dependent functions. The routines use the file **/usr/lib/terminfo** to describe what the terminal can do.

The routines are in the following categories:

- Screen updating
- Screen updating with user input
- Cursor motion optimization.

You can use motion optimization by itself. You can use screen updating and input without knowing about either motion optimization or the data.

New Terms

The extended curses routines use the concepts and terms listed in the following:

Term	Definition
terminal	Sometimes called ***terminal screen***, the ***terminal*** is a memory image of what the terminal screen currently looks like. This is a special screen.
screen	A ***screen*** is a special type of window that is as large as the terminal screen. You can define screens for a program. The extended curses routines define two screens for their use: **stdscr** The ***standard screen*** is a memory image of the screen that the routines make changes to. **curscr** The ***current screen*** is the actual image that is currently on the terminal.
window	A memory image of what a section of the terminal screen looks like at some point in time. A window can be either the entire terminal screen, or any smaller portion down to a single character.
Presentation Space	The data and attribute array associated with a window.
Pane	An area of the display that shows all or a part of the data contained in a presentation space associated with that pane. A pane is a subdivision of a panel.
Panel	A rectangular area on the display consisting of one or more panes that a program can treat as a unit. That is, the panes in a panel are displayed together, erased together and represent a unit to the operator. The routines stack or overlap panels on the screen, and remember the order of the stack and the contents of each panel.
Field	An area in a presentation space where the program can accept operator input.
extended character	A character that is represented in 2 bytes.
partial-character indicator	A character which replaces half a 2-byte, 2-column character when both bytes cannot be displayed or stored. The partial-character indicator is @ (at sign).

What You Need

To use the library, define the types and variables that the routines use. The file **cur01.h** contains all of the definitions that are needed for the library routines for most common uses. Include this file in the program by putting the statement:

```
#include <cur01.h>
```

at the top of the program source. When using the library routines for panel and pane management (those routines begin with the letters **ec**), use the statement:

```
#include <cur05.h>
```

in the program source to include a larger set of definitions.

Use an additional header file **cur00.h** in the program if the program uses the global variables defined to represent the information taken from the terminal file. This header file also contains include statements for the following header files:

- **stdio.h**
- **sgtty.h**
- **cur01.h**

You do not need to include those files separately in the program.

To compile a program with the **cc** command, specify the two additional libraries shown in the following example on the command line. This example compiles the program, `myprog.c`, with the linked output going to **a.out**, by using the following command:

```
cc myprog.c -lcur -lcurses
```

See Chapter 3, "Using the Subroutine Libraries" on page 3-1 for information about using libraries in a program.

Using the Screen Update Routines

To update the screen, the routines must know what the screen currently looks like and what it should be changed to. The routines define the **WINDOW** data type to hold this information. This data type is a structure that describes a window image to the routines, including the starting position on the screen (the (line, col) coordinates of the upper left corner) and size. See Appendix B, "Extended curses Structures" on page B-1 for a definition of the WINDOW structure.

A window is like an array of characters on which to make changes. Using the window, a program builds and stores an image of a portion of the terminal that it later transfers to the actual screen. When the window is complete, use one of the following routines to transfer the window to the terminal:

refresh Transfers the contents of **stdscr** to the terminal

wrefresh Transfers the contents of a named window (not **stdscr**) to the terminal

ecrfpl Transfers the contents of a named panel to the terminal

ecrfpn Transfers the contents of a named pane to the terminal.

This two-step process maintains several different copies of a window in memory and selects the proper one to display at any time. In addition, the program can change the contents of the screen in any order. When it has made all of the changes, the library routines update the terminal in an efficient manner.

What the Screen Looks Like

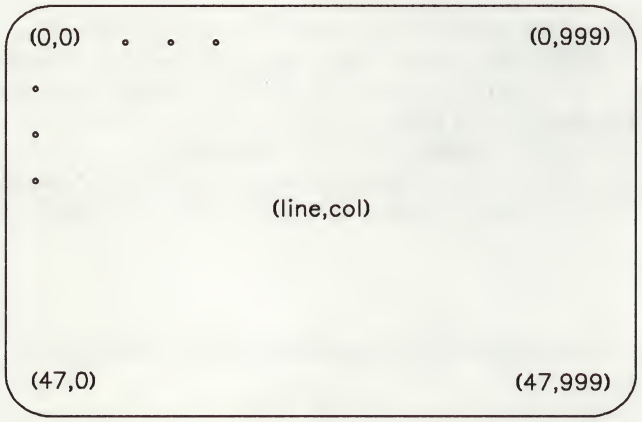
The screen is a matrix of character positions that can contain any character from the character set (see Appendix D, "ASCII Characters" on page D-1) that can be displayed. Do not use control characters except when the descriptions of the library indicate that you can. The actual dimensions of the matrix are different for each type of terminal. These dimensions are defined when the **initscr** routine calls the **terminfo** initialization subroutine, **setupterm**. For more information about **setupterm**, see *AIX Operating System Technical Reference*. However, the routines enforce the following limits on the terminal:

Coordinate	Description
lines	If the terminal specification defines less than 5 lines, the routines use a value of 24 lines.
columns	If the terminal specification defines less than 5 columns, the routines use a value of 80 columns.

Line 0 is at the top of the screen. Line values in the routine syntax are represented by `line`. Column 0 is at the left side of the screen. Column values in the routine syntax are represented by `col`. When used in calls to the library routines, the *line* value comes first.

```
move(line, col);
```

Figure 5-1 on page 5-8 shows the coordinate boundaries for a screen with 48 lines and 1000 columns.



A5AC6025

Figure 5-1. Screen Coordinate Boundaries

Function Names

Functions that change the contents of a specified window begin with **w**. Functions that change the contents of **stdscr** do not begin with **w**. For example, the function `addch` adds a character to **stdscr**, but the function `waddch` adds a character to a specified window. If you use a function with **w**, you must specify a window argument.

Note: Functions that do not have both a version with and without the **w** must specify a window.

Use the routines **move** and **wmove** to change the current (line, col) coordinates from one point to another. To move and then write to or read from the new position, use the following shorthand method for most routines:

1. Add the letters **mv** to the front of the routine name.
2. The first two arguments of the routine must be the (line, col) coordinates of the destination of the move.

For example, the following call sequence:

```
move(line, col);
addch(ch);
   .
   .
   .
wmove(win, line, col);
waddch(win, ch);
```

is the same as the following call sequence:

```
mvaddch(line, col, ch);
   .
   .
   .
mvwaddch(win, line, col, ch);
```

Note: The routines which use the **w** provide a window description pointer *win* before the standard routine arguments.

Variables

The following system variables defined in the header files describe the terminal environment. Use these variables in a program.

Name	Type	Description
My_term	bool	If this value is TRUE, the routines use the terminal type specified by Def_term as the terminal type being used. If this value is FALSE, the routines first check the terminal type specified by $TERM in the system environment. If $TERM is not specified, the routines use the value in Def_term.
Def_term	char*	Default terminal type if $TERM is not specified in the system environment.
COLS	int	Number of columns on the terminal.
ERR	int	Flag that the routines return when a failure occurs.
LINES	int	Number of lines on the terminal.
OK	int	Flag that the routines return when the function completes successfully.
curscr	WINDOW*	Current version of the terminal screen.
stdscr	WINDOW*	Standard screen.

The routines also define the following **#define** constants and types:

bool A type of *boolean* used as:

```
bool doneit;     /* defines variable doneit */
```

reg A type with storage class *register* used as:

```
reg int i;       /* defines i */
```

FALSE The value of boolean false (0).

TRUE The value of boolean true (1).

NLSCHAR A data type defined as unsigned short.

ATTR A data type defined as unsigned short.

Extended Characters

Extended characters require special consideration. An extended character requires 2 bytes of storage. It also occupies two columns on the screen. Routines which handle characters can accept extended characters as arguments, but generally cannot handle separate bytes of an extended character. Display attributes apply to entire characters. Cursor motion is constrained to character (rather than column) boundaries.

In a few circumstances, extended characters are not allowed as character arguments. For example, the borders of a box drawn around a window cannot be extended characters.

Various situations can arise when half an extended character must be replaced on the screen, in memory, or both. In these cases the remaining half of the character is replaced by the partial- character indicator. For example, if a window containing extended characters is obscured by another window and extended characters are partially covered, the visible halves are displayed as @ (at sign). The full character reappears when the covering window is removed.

Using the Library Routines

The following paragraphs outline the steps to follow when building a program to use these routines. The description uses routines that change **stdscr**, but the same concepts work with any window when using the **w** form of the routine as described in "Function Names" on page 5-9. See *AIX Operating System Technical Reference* for complete descriptions of these library routines, and their **w** forms.

Setting Up the Environment

To use these routines, the program must set up the operating conditions for the program. Perform the following actions in the program, if they apply, in the order that they appear in the following procedure:

1. Perform all necessary actions to load the program and make sure that it is operating successfully.
2. To change the defined size of the terminal, set the variables LINES and COLS to new values.
3. Use the routine **initscr** to get information about terminal characteristics, and to allocate memory for **stdscr** and **curscr**. Call **initscr** before calling any routines that affect windows. If the program uses a window routine before **initscr**, the program will not run.
4. Check the value that **initscr** returns to see if the screen setup was successful. If this value is 0 (FALSE or ERR), then **initscr** could not get enough memory for the needed windows.
5. Use any needed terminal status changing routine, such as **nl** or **crmode**.
6. Create any new windows with the **newwin** or **subwin** routine.
7. Create panels using **ecbpls**, **ecbpns**, **ecdvpl** or **ecdfpl**.
8. Define or change the characteristics of the windows as needed. For example, the routine **scrollok** allows the window to scroll, or the routine **leaveok** leaves the cursor at the position of the last change.

The program can now work with the windows that it has defined. When the program is done, use the routine **endwin** to clean up before exiting the program. This routine restores terminal modes to what they were when the program first started.

Writing to a Window

Use the following functions to change the contents of a window. Refer to *AIX Operating System Technical Reference* for complete information about each routine.

Routine **Description**

addch(c)
Adds the character `c` on the window at the current (line, col) coordinates.

waddfld
Adds a string to the window within a specified area (field) starting at the current coordinates.

addstr(str)
Adds the string pointed to by `str` on the window at the current (line, col) coordinates.

box(win, vert, hor)
Draws a box around the window using `vert` as the character for drawing the vertical sides, and `hor` for drawing the horizontal lines. Only single byte characters are allowed with `vert` and `hor`. See also **fullbox** and **cbox**.

cbox(win)
Draws a box around the window using the box characters defined in **/usr/lib/terminfo** (BX[]). If no box characters are defined, it uses | for vertical lines, - for horizontal lines, and + for corners. See also **box** and **fullbox**.

chgat(num_chars, mode)
Changes the attributes of the next `num_chars` characters, starting at the current (line, col) coordinates to the attribute(s) specified by `mode` (one or more of the attributes defined in "Display Attributes" on page 5-27). See also **pchgat**.

clear
Resets the entire window to blanks.

clearok(scr, boolf)
Sets the clear flag for the screen `scr` to the value of `boolf`.

clrtobot
Clears the window from the current (line, col) coordinates to the bottom.

clrtoeol
Clears the window from the current (line, col) coordinates to the end of the line.

colorend
Returns the terminal to *normal* attributes following a **colorout** call.

Routine	Description
colorout(mode)	Sets the current standout bit pattern (_csbp in the **window** structure) to the value of `mode` (one or more of the attributes defined in "Display Attributes" on page 5-27) and turns on **_STANDOUT**. All characters following this call are displayed with `mode` as the attribute.
delch	Deletes the character at the current (line, col) coordinates.
deleteln	Deletes the current line.
ecactp	Specifies the active pane.
ecshpl	Shows a specified panel by bringing it to the top of the stack of panels.
ecrfpl	Refreshes the panel on the display.
ecrfpn	Refreshes the pane on the display.
ecrmpl	Removes a panel from the display.
ecscpn	Scrolls a specified pane.
erase	Erases the window to blanks without setting the clear flag. See also **perase**.
fullbox(win, vert, hor, topl, topr, botl, botr)	Draws a box around the window using `vert` as the character for vertical sides, `hor` for horizontal, and `topl`, `topr`, `botl` and `botr` as the corner characters. The border characters must all be 1-byte characters. See also **box** and **cbox**.

Routine	Description
insch(c)	Inserts character `c` at the current (line, col) coordinates.
insertln	Inserts a line above the current line.
move(line, col)	Changes the current (line, col) coordinates of the window to (line, col).
overlay(win1, win2)	Overlays `win1` on `win2`. Windows need not be the same size.
overwrite(win1, win2)	Overwrites `win1` on `win2`. Windows need not be the same size.
printw(fmt, arg1, arg2, ...)	Performs a **printf** on the window starting at the current (line, col) coordinates.
refresh	Writes the contents of the specified window to the terminal.
standend	Stops putting characters onto `win` in standout mode.
standout	Starts putting characters onto `win` in standout mode.

Use the **refresh** routine to transfer the contents of the current window to the screen after all changes to the window are complete. The **refresh** routine does not rewrite any part of the window that has not changed since the last refresh call. To force the whole window to be rewritten, use the **touchwin** routine before the **refresh** routine. Also use **ecrfpn** to refresh a pane, and **ecrfpl** to refresh a panel.

Getting Input from the Terminal

Input is the complementary function to output. The screen package needs to know what is on the terminal at all times. Therefore, if a program echoes input characters, the terminal must be in a mode that passes characters immediately to the program, rather than waiting for a carriage return to send input to the program. The **getch** routine sets the terminal to the character input mode and then reads in the character.

Use the following routines for input from the terminal:

Routine **Description**

crmode
Sets the terminal to allow character by character input and not wait for a carrier return to send input to the process.

nocrmode
Sets the terminal to wait for a carrier return to send input to the process.

echo
Sets the terminal to echo characters.

noecho
Sets the terminal to not echo characters.

ecflin
Gets input from the screen as long as the cursor is in a specified area (***field***) of the screen.

ecpnin
Gets input from the screen in a specified pane, and scrolls the pane as needed to keep the cursor in the field.

extended ()
Controls international character support input processing.

getch
Gets a character from the terminal and (if necessary) echoes it on the window.

getstr(str)
Gets a string through the window and puts it in the location pointed to by `str`. The location must be large enough to hold the string. The string is terminated by `\n` (new-line).

Routine	Description
keypad	Controls the mapping of input from the keyboard. See "Controlling Input with the keypad, extended, and trackloc Routines" on page 5-33 for more information.
raw	Sets the terminal to raw mode.
noraw	Resets the terminal from raw mode.
scanw(fmt, arg1, arg2, ...)	Performs a **scanf** through the window using `fmt`.

Controlling the Screen

Use the following library routines to control and manipulate the windows, panes, and panels on the screen.

Routine	Description
delwin(win)	Deletes the window and frees the resources assigned to the window.
ecadpn	Adds the specified window to the list of windows that can be displayed in a pane, but does not display it.
ecaspn	Specifies a window to be displayed in the specified pane, but requires a **refresh** call to display it.
ecbpls	Builds a panel structure.
ecbpns	Builds a pane structure.
ecdfpl	Creates **WINDOW** structures to define a panel.
ecdppn	Removes the specified window from the list of windows that can be displayed in the pane.

Routine	Description
ecdspl	Returns all structures associated with a panel to the storage pool, including structures for panes linked to the panel.
ecdvpl	Divides a panel into panes. All panes must be defined, and be linked to the panel.
ecrlpl	Returns structures associated with a panel to the storage pool, but not those that define the panel or the panes linked to the panel.
endwin	Restores the terminal to the state it was before **initscr** was called. Always use **endwin** before exiting.
gettmode	Gets the information about the terminal. This routine is called by **initscr**.
getyx(win, line, col)	Puts the current (line, col) coordinates of `win` in the variables `line` and `col`.
inch	Returns the character at the current (line, col) coordinates on the specified window.
initscr	Initializes the screen routines. Call this routine before using any of the screen routines. Use the **endwin** before exiting the screen routines.
leaveok(win, boolf)	Sets the boolean flag **_leave** to the value specified by `boolf`. This flag indicates that the cursor should be positioned after the last change.
longname	Return the long (full) name of the terminal described by the **terminfo** entry.

Routine Description

mvcur(lastline, lastcol, newline, newcol)
Moves the terminal cursor from (`lastline, lastcol`) to (`newline, newcol`).

Note: Each window and the terminal have a cursor. The terminal cursor becomes the cursor on the active window or pane.

mvwin(win, line, col)
Moves the home position of the window `win` from its current starting coordinates to (line, col).

newview(orig_win, num_lines, num_cols)
Creates a new window that is `num_lines` lines and `num_cols` columns. The window is a viewport of the `orig_win` starting at the current (line, col) coordinates of `orig_win`. A viewport of a window containing extended characters should include the entire width of the original window.

newwin(lines, cols, begin_line, begin_col)
Creates a new window with `lines` lines and `cols` columns starting at position (`begin_line, begin_col`).

nl
Sets new-line mode so that the system starts changing return characters to linefeed characters.

nonl
Resets new-line mode so that the system does not change return characters. This setting helps the **refresh** routine perform optimization.

resetty
Restores the tty characteristic flags to what **savetty** stored.

savetty
Saves the current tty characteristic flags.

scroll(win)
Scrolls the window upward one line.

scrollok(win, boolf)
Sets the scroll flag for the given window to the value specified by `boolf`. A value of FALSE (disable scrolling) is the default setting.

Routine Description

setterm(name)
Sets the terminal characteristics to be those of the terminal name `name`.

subwin(win, lines, cols, begin_line, begin_col)
Creates a new window with `lines` lines and `cols` columns starting at position (`begin_line, begin_col`) in the window `win`.

touchwin(win)
Forces the **refresh** routine to write all of the specified window, instead of just the parts that have changed.

trackloc
Controls the tracking of the locator cursor.

tstp
When using the tty driver, this function saves the current tty state and then puts the process to sleep. When the process is started again, the process restores the tty state and then calls **wrefresh(curscr)** to redraw the screen. The **initscr** routine sets the signal SIGTSTP to trap to this routine.

unctrl(ch)
Returns a string which is a representation of `ch`. To use **unctrl**, put the statement:

```
#include <cur04.h>
```

in the program file.

vscroll(view_win, deltaline, deltacol)
Scrolls the viewport window (see **newview**) down `deltaline` lines and right `deltacol` columns. If the numbers are negative, the directions are up and left, respectively.

Routines for Panels and Panes

The extended curses library contains routines to help create a screen appearance similar to that used for Usability Services. The following paragraphs describe the concept of the panel and pane interface, and list the routines for creating a panel and pane interface. Refer to *AIX Operating System Technical Reference* for detailed information about each routine.

Defining Panels and Panes

To define a panel, provide the following information about the panel:

- The size of the panel as it appears on the display
- The location on the display of the upper left corner of the panel
- Whether the panel is to have a border or not
- How the panel is to be divided into panes.

In addition, provide the following information for each pane within the panel:

- The size of the presentation space associated with the pane
- The relative size of the pane within the panel
- Whether the pane is to have a border
- If and how the pane is to be further divided into smaller panes.

To divide panels and panes into smaller panes, follow a few simple rules. These rules ensure that a program can access all areas on the panel or pane that it creates:

- You can divide a panel or pane either horizontally (using a horizontal dividing line) or vertically (using a vertical dividing line).
- Panes created by a horizontal division must be linked together from top to bottom.
- Panes created by a vertical division must be linked together from left to right.
- Panes that are divided again must be linked to the first pane of its subpanes. The original pane in this case is not a part of the presented panel, but it is needed to define the structure of the panel.
- Panes created by a horizontal division have a fixed horizontal dimension that is the same as its parent pane.
- Panes created by a vertical division have a fixed vertical dimension that is the same as its parent pane.
- Specify the variable dimension for a pane as being in one of three categories:

 Fixed For a ***fixed*** pane, specify the number of rows or columns, including any border, to assign to the pane.

 Fractional For a ***fractional*** pane, specify the percentage of the available space to assign to the pane.

 Floating For a ***floating*** pane, do not specify a size. The floating pane shares the available space equally with any other floating panes that the program creates.

The linkage of the panes forms a tree structure. The root of the tree is a panel description. All other elements in the tree are pane descriptions.

Creating Panels and Panes

To create a panel that looks like the outline shown in Figure 5-2 on page 5-24, perform the following steps (following the rules for dividing panels and panes) as shown in Figure 5-3 on page 5-24.

1. Define panel **P** using the **ecbpls** routine with a link to pane **A**.
2. Divide the panel (**P**) with two horizontal splits into three panes. Use the **ecbpns** routine to define the three panes with the following links:

A	No links
B	Linked to **A** and **D**
C	Linked to **B** and **F**

3. Divide pane **B** with a single vertical split into two panes. Use the **ecbpns** routine to define the two panes with the following links:

D	No links
E	Linked to **D**

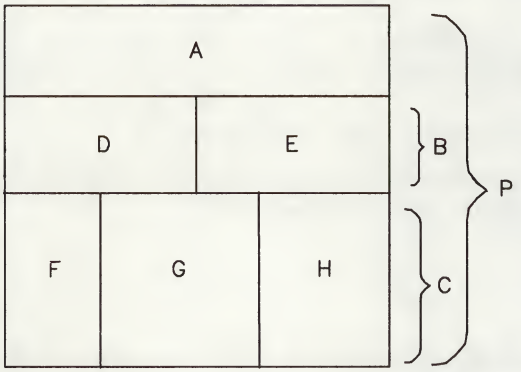
4. Divide pane **C** with two vertical splits into three panes. Use the **ecbpns** routine to define the three panes with the following links:

F	No links
G	Linked to **F**
H	Linked to **G**

Although the program must create panes **B** and **C** to get the smaller panes, those two panes do not appear as panes in the final display.

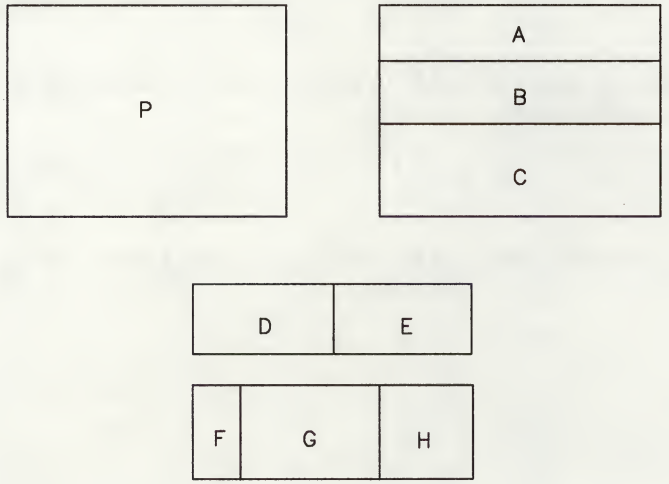
Figure 5-4 on page 5-25 shows how the panel and pane descriptions for the final structure are linked. Horizontal lines show the links within a pane; vertical lines show links to the parent panel or pane.

The program in Figure 5-5 on page 5-26 creates the panel shown in Figure 5-2 on page 5-24.



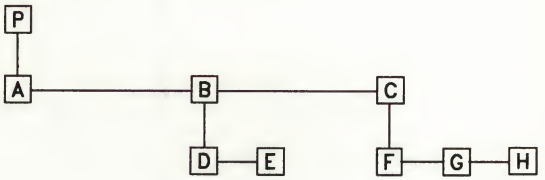
A5AC6026

Figure 5-2. Example Panel Final Appearance



A5AC6022

Figure 5-3. Creating Panes in the Panel



A5AC6028

Figure 5-4. Links in the Panel and Pane Structure

```
#include <cur01.h>
#include <cur05.h>

main()

{

      PANE    *A, *B, *C, *D, *E, *F, *G, *H ;
      PANEL   *P ;

initscr () ;

A = ecbpns (24, 80, NULL, NULL, 0, 2500, Pdivszp, Pbordry, NULL, NULL);

D = ecbpns (24, 80, NULL, NULL, 0, 0, Pdivszf, Pbordry, NULL, NULL);
E = ecbpns (24, 80, D,    NULL, 0, 0, Pdivszf, Pbordry, NULL, NULL);

B = ecbpns (24, 80, A, D, Pdivtyh, 3000, Pdivszp, Pbordry, NULL, NULL);

F = ecbpns (24, 80, NULL, NULL, 0, 0,    Pdivszf, Pbordry, NULL, NULL);
G = ecbpns (24, 80, F,    NULL, 0, 5000, Pdivszp, Pbordry, NULL, NULL);
H = ecbpns (24, 80, G,    NULL, 0, 3000, Pdivszp, Pbordry, NULL, NULL);

C = ecbpns (24, 80, B, F, Pdivtyh, 0, Pdivszf, Pbordry, NULL, NULL );

P = ecbpls (24, 80, 0, 0, NULL, Pdivtyv, Pbordry, A );

ecdvpl ( P );
ecdfpl ( P, FALSE );
ecshpl ( P );
ecrfpl ( P );

endwin();
}       /* end main program */
```

Figure 5-5. Program to Create Example Panel

Display Attributes

Use the color and display characteristics defined in Figure 5-6 on page 5-28. These names are external variables that define the attributes that a program can use on the current terminal. The values of these variables depend on the capabilities of the current terminal and the priorities that you assign to the attributes. Change the values of these variables with the **sel_attr** routine as explained in "Changing the Defined Attributes" on page 5-29.

The characteristics that a program selects for the terminal are loaded into the attribute variable (of data type ATTR) associated with the data being displayed. Select as many of the attributes as needed, but those selected are packed into the attribute variable (of data type ATTR) in the following order:

1. REVERSE,
2. INVISIBLE,
3. F_WHITE,
4. F_RED,
5. F_BLUE,
6. F_GREEN,
7. F_BROWN,
8. F_MAGENTA,
9. F_CYAN,
10. F_BLACK,
11. B_BLACK,
12. B_RED,
13. B_BLUE,
14. B_GREEN,
15. B_BROWN,
16. B_MAGENTA,
17. B_CYAN,
18. B_WHITE,
19. BOLD,
20. UNDERSCORE,
21. BLINK,
22. DIM,
23. STANDOUT,
24. PROTECTED,
25. FONT0,
26. FONT1,
27. FONT2,
28. FONT3,
29. FONT4,
30. FONT5,
31. FONT6,
32. FONT7,
33. NULL.

To change the order, see "Changing the Defined Attributes" on page 5-29. Once the attribute variable (of data type ATTR) is full, the routines ignore the remaining lower priority attributes. If an attribute does not work with the current display, the routines ignore that attribute. Therefore, you can specify color attributes and still be able to use the program with a monochrome display. Figure 5-6 defines the external variable names that the routines use to set the display attributes.

Name	Attribute
UNDERSCORE	Display characters with underline.
REVERSE	Display characters in reverse video.
NORMAL	Display characters without highlighting (return to normal).
INVISIBLE	Do not display characters.
STANDOUT	Display characters in high intensity (can be used with other attribute colors). On many terminals, this is the same as REVERSE.
BOLD	Display characters in bold font (or high intensity on some terminals).
BLINK	Display blinking characters (can be used with other attribute colors).
DIM	Display characters in reduced intensity.
PROTECTED	Protected display field.
F_BLACK	Set foreground color to black.
F_BLUE	Set foreground color to blue.
F_GREEN	Set foreground color to green.
F_CYAN	Set foreground color to cyan.
F_RED	Set foreground color to red.
F_MAGENTA	Set foreground color to magenta.
F_BROWN	Set foreground color to brown.
F_WHITE	Set foreground color to white.
B_BLACK	Set background color to black.
B_BLUE	Set background color to blue.
B_GREEN	Set background color to green.
B_CYAN	Set background color to cyan.
B_RED	Set background color to red.
B_MAGENTA	Set background color to magenta.
B_BROWN	Set background color to brown.
B_WHITE	Set background color to white.
FONT0	Select defined character font 0.
FONT1	Select defined character font 1.
FONT2	Select defined character font 2.
FONT3	Select defined character font 3.
FONT4	Select defined character font 4.

Figure 5-6 (Part 1 of 2). Display Attributes

Name	Attribute
FONT5	Select defined character font 5.
FONT6	Select defined character font 6.
FONT7	Select defined character font 7.

Figure 5-6 (Part 2 of 2). Display Attributes

Changing the Defined Attributes

To change the characteristics assigned to the external variables listed in Figure 5-6 on page 5-28, use the **sel_attr** routine. This routine uses a set of defined constants contained in the header file **cur03.h**. To use this routine, put the following statement at the beginning of the program file:

```
#include <cur03.h>
```

The file **cur03.h** defines the following constants:

_dNORMAL
_dREVERSE
_dBOLD
_dBLINK
_dUNDERSCORE
_dDIM
_dINVISIBLE
_dPROTECTED
_dSTANDOUT
_dF_BLACK
_dF_RED
_dF_GREEN
_dF_BROWN
_dF_BLUE
_dF_MAGENTA
_dF_CYAN
_dF_WHITE
_dB_BLACK
_dB_RED
_dB_GREEN
_dB_BROWN
_dB_BLUE
_dB_MAGENTA
_dB_CYAN
_dB_WHITE
_dFONT0
_dFONT1
_dFONT2
_dFONT3
_dFONT4
_dFONT5
_dFONT6
_dFONT7

These constants are only valid when using the **sel_attr** routine. They cannot be used with any other routine.

Changing Screen Attributes

The code fragment in Figure 5-7 shows how to use these constants to change the default set of attributes.

```
#include <cur00.h>
#include <cur03.h>

int     attrs[] =
{
        _dBOLD, _dBLINK,
        _dF_WHITE, _dF_RED, _dF_BLUE, _dF_GREEN,
        _dF_BROWN, _dF_MAGENTA, _dF_CYAN, _dF_BLACK,
        _dB_BLACK, _dB_RED, _dB_BLUE, _dB_GREEN,
        _dB_BROWN, _dB_MAGENTA, _dB_CYAN, _dB_WHITE,
        _dREVERSE, _dINVISIBLE, _dDIM, _dUNDERSCORE,
        NULL
};

main( )
{

sel_attr(attrs);
initscr( );
if( REVERSE == NORMAL ) REVERSE = F_BLACK | B_WHITE;
if( INVISIBLE == NORMAL ) INVISIBLE = F_BLACK | B_BLACK;
if( DIM == NORMAL ) DIM = F_BLACK | BOLD;
if( UNDERSCORE == NORMAL ) UNDERSCORE = F_WHITE | B_RED;
STANDOUT = REVERSE;

        <rest of program>

endwin( );
}  /* end main */
```

Figure 5-7. Changing Screen Attributes Example Program

The routines define 16 bits of unique attribute information. Selecting foreground color, background color or font requires either 1, 2 or 3 bits depending upon the number of colors or fonts in the list: 1 bit for 2 or fewer, 2 bits for 3 or 4, and 3 bits for 5 to 8. Each character attribute takes 1 bit. However, the attribute names passed to **wcolorout** are variables, so that you can make combinations from the other attributes as shown in the last part of the previous example. If a requested attribute (that is not the terminal default) is equal to NORMAL, then it is either not supported by the terminal, or there is not enough space in the attribute variable (of data type ATTR) for its mask.

Using Other Features

Controlling Input with the keypad, extended, and trackloc Routines

The **keypad** routine allows a program to recognize control sequences in the input without searching the input or introducing device dependencies. If **keypad** is active, it scans all input data for control sequences. If it finds a control sequence, it returns the associated code to the program instead of the actual control sequence. The control codes are shown in Figure 5-8. These codes are defined in the file **cur02.h** with values greater than 0x100.

To get international character support processing on input, **extended** must be active. If **extended** is turned off, shift codes and data codes input separately.

To get tracking of the locator cursor on the screen, **trackloc** must be active. If **trackloc** is turned off, the application has to handle the tracking of the locator cursor. For information about locator input, see *AIX Operating System Technical Reference.*

Name	Description
KEY_NOKEY	No keyboard data and no delay on
KEY_BREAK	Break
KEY_DOWN	Cursor down
KEY_UP	Cursor up
KEY_LEFT	Cursor left
KEY_RIGHT	Cursor right
KEY_HOME	Home - top left
KEY_BACKSPACE	Backspace
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert character mode start
KEY_EIC	Exit insert character mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll forward

Figure 5-8 (Part 1 of 3). Control Codes

Name	Description
KEY_SR	Scroll backward (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab stop
KEY_CTAB	Clear tab stop
KEY_CATAB	Clear all tab stops
KEY_ENTER	Enter key
KEY_SRESET	Soft reset key
KEY_RESET	Hard reset key
KEY_PRINT	Print or copy
KEY_LL	Lower left (last line)
KEY_A1	Pad upper left
KEY_A3	Pad upper right
KEY_B2	Pad center
KEY_C1	Pad lower left
KEY_C3	Pad lower right
KEY_DO	DO key
KEY_QUIT	QUIT key
KEY_CMD	Command key
KEY_PCMD	Previous command key
KEY_NPN	Next pane key
KEY_PPN	Previous pane key
KEY_CPN	Command pane key
KEY_END	End key
KEY_HLP	Help key
KEY_SEL	Select key
KEY_SCR	Scroll right key
KEY_SCL	Scroll left key
KEY_TAB	Tab key
KEY_BTAB	Back tab key
KEY_NEWL	New-line key
KEY_F0	Function key - 128 values
KEY_F(n)	Not used
KEY_ESC1	Added to the ending character code for ESC sequences in the form ESC c with c in the range 0x30 - 0x7f. The value sent is in the range 0x200 to 0x24f.
KEY_ESC2	Added to the ending character code for ESC sequences in the form ESC [s c with c in the range 0x40 - 0x7f. The value sent is in the range 0x250 to 0x28f.

Figure 5-8 (Part 2 of 3). Control Codes

To use the control sequences in a program, first use a call to the **keypad** routine:

```
keypad(TRUE);
```

Scrolling Windows

If a program defines a window with **scrollok**, scrolling is allowed for that window. When the program writes a character to the lower right corner of the window (or adds \n to the last row of the window), the routines automatically call the **scroll** routine.

If a program does not define the window with **scrollok**, scrolling is not allowed for that window. When the program writes a character to the lower right corner of the window (or adds \n to the last row of the window), the routines reset the current `col` coordinate to zero (beginning of line) and the window does not scroll.

If a window includes the lower right corner of the terminal screen, the flag byte bit, **_SCROLLWIN**, in the window structure for that window is set. This bit indicates that if a character is written to the lower right corner of the window, the terminal (*not* the routines) inserts a blank line at the bottom of the screen (scrolls) to make room for more information. To avoid this hardware scroll, the routines operate differently for a window that includes the lower right corner of the terminal screen and is not defined with **scrollok**. When a program writes a character to the lower right corner of such a window (and screen), the character is added to the data array for the window but the character is not written to the window when the window is refreshed.

To move a window to the lower right corner, use the **mvwin** routine. The **_SCROLLWIN** flag bit for that window is not automatically set. However, the **wrefresh** routine handles that window as if the **_SCROLLWIN** flag bit were set.

Improving Performance

To speed up output, create an output buffer using statements similar to the following program fragment:

```
#include  <stdio.h>

char      obuf[BUFSIZ];

main( )
{
   setbuf (stdout, obuf);
      .
      .
      .
   /* rest of program */
}
```

Example Program

Figure 5-9 on page 5-37 shows the use of some of the routines to create a series of displays on the screen. If you are using the Programming Examples, the program is stored as **twinkle.c**. Compile and run the program to see the effects of the Extended curses functions. To compile the example program, use the following command:

```
cc twinkle.c -lcur -lcurses
```

```
#include        <curOO.h>
#include <signal.h>

#define NCOLS 80
#define NLINES 24
#define MAXPATTERNS 11

struct locs
{
   char  y, x;
};

typedef struct locs  LOCS;

LOCS  layout[ NCOLS * NLINES ];  /* current board layout */

int   pattern,                   /* current pattern number */
      numstars;                  /* numbers of stars in ptern */
```

Figure 5-9 (Part 1 of 6). Example of Extended curses Program

```
main()
{
   char *getenv();
   int die();

   srand( getpid() );            /* initialize random sequence */
   initscr();
   signal( SIGINT, die );
   noecho();
   leaveok( stdscr, TRUE );
   scrollok( stdscr, FALSE );

   for( ;; )
   {
      makeboard();               /* make the board setup */
      puton( '*' );              /* put on '*'s */
      system( "sleep 2" );
      erase();
      refresh();
   }
}

die()
{
   signal( SIGINT, SIG_IGN );
   mvcur( LINES/2, COLS/2, 0, 0 );
   wclear( curscr );
   wrefresh( curscr );
   endwin();
   exit(0);
}
```

Figure 5-9 (Part 2 of 6). Example of Extended curses Program

```
makeboard()
{
   reg int  y, x;
   reg LOCS  *lp;

   pattern = rand() % MAXPATTERNS;
   lp = layout;
   for( y = 0; y < NLINES; y++ )
   {
      for( x = 0; x < NCOLS; x++ )
      {
         if( ison( y, x ))
         {
            lp -> y = y;
            lp++ -> x = x;
         }
      }
   }
   numstars = lp - layout;
}

ison( y, x )

reg int y, x;

{
   switch( pattern )
   {
      /*
      ** Alternating lines:
      */
      case 0:
         return !( y & 01 );
```

Figure 5-9 (Part 3 of 6). Example of Extended curses Program

```
/*
** Box:
*/
case 1:
   if( y < 3 || y >= NLINES - 3 )
      return TRUE;
   return( x < 4 || x >= NCOLS - 4 );
/*
** Cross:
*/
case 2:
   return( ( x + y ) & 01 );
/*
** Bar across center:
*/
case 3:
   return( y >= 9 && y <= 15 );
/*
** Alternating columns:
*/
case 4:
   return !( x & 02 );
/*
** Bar down center:
*/
case 5:
   return( x >= 36 && x <= 44 );
/*
** Bar across and down center:
*/
case 6:
   return( ( y >= 9 && y <= 15 ) || ( x >= 37 && x <= 43 ));
```

Figure 5-9 (Part 4 of 6). Example of Extended curses Program

```
/*
** Bar across and down center, in a box:
*/
case 7:
   if( y < 3 || y >= NLINES - 3 )
      return TRUE;
   if( x < 4 || x >= NCOLS - 4 )
      return TRUE;
   return( ( y >= 10 && y <= 14 ) || ( x >= 36 && x <= 44 ));
/*
** Asterisk:
*/
case 8:
   if( abs( x - y ) <= 2 || abs( NLINES - ( x + y )) <= 2 )
      return TRUE;
   if( abs( ( NLINES/2 ) - x ) <= 2 )
      return TRUE;
   return( abs( ( NLINES/2 ) - y ) <= 1 && x <= NLINES );
/*
**  Ellipse:
*/
case 9:
   return
   (
      (
         (( float ) (( x-40 ) * ( x-40 )) ) / 1521 +
         (( float ) (( y-12 ) * ( y-12 )) ) / 121
      ) <= 1
   );
```

Figure 5-9 (Part 5 of 6). Example of Extended curses Program

```
      /* Circle: */
      case 10:
         return
         (
            (
               (( float ) (( x-28 ) * ( x-28 )) ) / 729 +
               (( float ) (( y-12 ) * ( y-12 )) ) / 121
            ) <= 1
         );
   }     /* end of switch( pattern ) */
}        /* not reached */

puton(ch)
reg char ch;
{
   reg LOCS  *lp;
   reg LOCS  *end;
   LOCS      temp;
   reg  int     r;

   end = &layout[ numstars ];
   for( lp = layout; lp < end; lp++ )
   {
      r = rand() % numstars;
      temp = *lp;
      *lp = layout[ r ];
      layout[ r ] = temp;
   }
   for( lp = layout; lp < end; lp++ )
   {
      mvaddch( lp -> y, lp -> x, (NLSCHAR) ch );
      refresh();
   }
}  /* end of twinkle */
```

Figure 5-9 (Part 6 of 6). Example of Extended curses Program

Chapter 6. Writing Messages and Help

CONTENTS

About This Chapter 6-2
Messages 6-3
 Message Format 6-4
Building a Message Table 6-8
 Copying the Standard Format File 6-8
 Naming the Message Table 6-10
 Adding Message Definitions 6-10
 Message Index 6-11
 Adding Text Insert Definitions 6-11
Using Messages in a Program 6-13
 Including Header Files 6-13
 Using Routines to Display Messages 6-14
Using Variable Fields in Message Text 6-15
 Example of the Integer Symbol 6-16
 Example of the Long Integer Symbol 6-18
 Example of the Character String Symbol 6-18
 Example of Text Insert Symbol 6-20
Help 6-22
 Help Format 6-22
 File Path Name 6-23
 Changing the File Path Name 6-23
 Changing the File Path Name for Debugging 6-24
Building a Help File 6-25
 Content of the Help Text File 6-25
Using Help in a Program 6-29
 Including Header Files 6-29
 Using Routines to Display Help 6-29

About This Chapter

The programs that run on the RT work station should provide feedback information to the person using the program. If an error occurs while the program is running, whether the error is produced by the program or the operator, provide an indication to the operator that the error occurred. Error indications, or any brief information that a program writes to standard error or a queue, are called ***messages***.

In addition, a program may provide information that explains in detail how the program operates. This information can include command summaries, operating procedures, explanation of ideas or information to make using the program easier to understand. Explanatory information that a program provides is called ***help***.

This chapter explains how to use the operating system services to provide both messages and help from a program. It describes the message and help text files, how to make them, and how to incorporate them into a program. The chapter also describes the format of messages.

Messages

A ***message*** is information that the program generates to inform the person using the program, or **/dev/console** of conditions in the program. If the conditions require steps to recover, the message provides those steps. A program can generate two types of messages:

Immediate message
: The message appears on the screen associated with the program. The message is usually in response to something that the person using the program did.

Queued message
: The message appears in the message queue file **/qmsg** and can only be seen by listing or editing the message queue file. When a message enters the queue file, a beep tone notifies **/dev/console**. Programs that operate directly with the user usually do *not* produce queued messages. Background processes, such as daemons, produce queued messages.

The operating system provides a set of routines, called ***message services*** to help create, update, and display messages from a program. The routines are in the library file **/usr/lib/librts.a**. The services for generating messages include:

- A standard message format that matches the format of the operating system messages
- A file containing a template to use to create messages
- Two routines that help to generate either immediate or queued messages from a program
- Header files to simplify declarations needed to use message services
- Variable field symbols in the messages. When message services displays the message, it replaces these symbols with values that you specify.

Message Format

Each of the two types of messages has a different format. When displayed, queued messages are in the following format:

```
MM/DD  HH:mm  y       z  pgm-nnn
This is a sample queued message.
```

Immediate messages have the following format.

```
pgm-nnn This is a sample immediate message:
                a. Short line.
                b. This is a long line.  Note that
                   the message is presented
                   as you format it. It is not
                   reformatted before being displayed.
        Time = HH:mm.  Severity = y.  Error Number = z.
```

The symbols have the following meaning. See "Using Routines to Display Messages" on page 6-14 for a description of the library routines for generating messages.

Field	Description
pgm	A 3-character program identifier that is unique to the program. To ensure that these characters do not conflict with numbers already assigned to system programs, choose a number that is larger than 500. You can also use alphabetic characters for the program identifier. Figure 6-2 on page 6-6 shows some of the identifiers that the system programs use. To match the style of the system messages, choose three digits for the program identifier. Message services displays this identifier to help the operator know what program generated the message. Message services does not use the identifier as an index into the messages.
nnn	A 3-digit sequence number for the message within the set of messages for the program identifier specified by *pgm*. This number allows looking up a description of the message in a book. Message services does not use the number.

Figure 6-1 (Part 1 of 2). Message Fields

Field	Description
text	The words that explain the condition associated with the message.
MM/DD	The month and day that the message was generated.
HH:mm	Time (24 hour format) that the message was issued. When a program generates an immediate message with the **msgimed** function and uses the **msgfltim** flag, message services supplies the time. Message services automatically provides the time for queued messages.
y	Severity code is the severity code specified when using the message services routines to write the message. If you do not specify a severity code, this field does not appear.
z	Error number is the error code specified when using the message services routines to write the message. If you do not specify an error code, this field does not appear.

Figure 6-1 (Part 2 of 2). Message Fields

For example, if the operator makes a mistake when entering the date, the program could generate the following message:

```
345-007 The system cannot recognize the date that you
        entered.  Please enter the date again.
```

The optional information (time, severity or error code) does not appear in this message because the program did not specify that they be displayed when it called for the message. The book for this program should include an entry for error number 345-007 that contains information about the correct date format, or other information to help correct this problem.

ID	System Program
000	Common
001	BASIC Compiler
002	BASIC Interpreter
003	VRM ATOC
005	Coprocessor Configuration
006 - 007	Not Available
008	Multiple Work Station Install
009 - 012	Not Available
013	Install and Maintenance diskette
014 - 016	Not Available
017	Device Driver - APA8 Display
018 - 020	Not Available
021	Operating System Trace Points
022	Coprocessor Trace Points
023	Coprocessor Control
024 - 026	Not Available
027	VRM Dump
028 - 031	Not Available
032	VRM Debugger
033	Coprocessor Software
040 - 041	Operating System Configuration
042	Data Management Services
043	Data Management Utilities
044	Not Available
046	VRM Install
047	VRM Device Driver - Diskette
048	Install and Update Services
049 - 051	Not Available
052	VRM Device Driver - Streaming Tape
053 - 060	Not Available
061	Dialog Manager
062	Interactive Work Station
063	Not Available
064	Virtual Terminal Resource/Screen Control
065 - 067	Not Available
068	Device Driver - Keyboard
069	Activity Manager
070 - 074	Not Available

Figure 6-2 (Part 1 of 2). System Identifiers

ID	System Program
075	Hardware Access Support
076	Base LAN Install
077	Pascal Compiler
078	Not Available
079	Device Drivers - ports
080 - 081	Not Available
082	Print I/O Services
083 - 089	Not Available
090	Message Services
091	Tools Application
092 - 094	Not Available
095	Operating System
096	Dialog Definition Statements
097	Files Application
098	Not Available
099	Device Driver - Sound
100	Device Driver - Locator (mouse)
101 - 102	Not Available
103	Operating System
104 - 105	Not Available
106	Data Base Command Line
107	Data Base Program Interface
108	**dumpfmt** Command
109	Error Log
110	Trace
111 - 499	Not Available
500 +	Available for new programs

Figure 6-2 (Part 2 of 2). System Identifiers

Building a Message Table

The services of the operating system can help to build a table of messages that is separate from the source code of the program. To build the table of messages, first get a file containing the standard message format from the file system. Then add messages to that file, compile the table of messages (using **cc**), and link the messages with the compiled program (object modules) and the messages library **/usr/lib/librts.a**. This method keeps the messages in memory when the program is in memory. Do not use this method for long text, such as help (see "Help" on page 6-22).

Having a separate table of messages makes it easier to change messages, add messages, and translate the messages to another language.

Perform the following steps to build a message table and incorporate it in the program. Refer to the following paragraphs for additional explanation for some of the steps:

1. Copy the example message table file **msg07.h** into the current directory:

   ```
   cp /usr/include/msg07.h .
   ```

2. Rename the example file to the name of the message source file. Use a **.c** file extension.

   ```
   mv msg07.h mymsgs.c
   ```

3. Replace the name of the table, *tablename* in the example file, with the external name of the table.

4. Use an editor to add the message definitions and text to the message table source file.

5. Compile the message table and program source files using the **cc** command.

   ```
   cc program-files.c mymsgs.c -o myname
   ```

 In this command, *program-files.c* can be any number of C language source files each with a *.c* extension. The resulting executable program is in the file *myname*.

Copying the Standard Format File

In the directory **/usr/include** is a file called **msg07.h**. This file is an ASCII file that contains the framework for building a message table. Figure 6-3 on page 6-9 shows the major parts of this file.

```
                    /*******************/
 #define TABLE_NAME /***/ tablename /***/
                    /*******************/

 #include <msg08.h>
                     /* structure declarations */
/*
**          ** MESSAGE DEFINITIONS **
*/
 static msg__msg msg_defs[] = {

 0, "345", "007",
 "The system cannot recognize the date that you \n\
  entered.  Please enter the date again.",
                                  /* message 001 */
 } ;
/*
**          ** TEXT INSERT DEFINITIONS **
*/
 static msg__ins ins_defs[] = {

 "month",                         /* insert 001 */
 "day",                           /* insert 002 */
 "year",                          /* insert 003 */
 } ;

 #include <msg09.h>  /* pointers table */
```

Figure 6-3. Content of Message Standard Format File

Naming the Message Table

The name of the table is the name assigned to it in the first line of the table file. The following line defines a table name of `xyztab1`.

```
#define TABLE_NAME xyztab1
```

For consistency with system table conventions, use the following guidelines when naming the table:

1. The first three characters should be the program identifier (`xyz` in the example).
2. The next three characters should be the letters `tab` to indicate a table.
3. The last character should be an identifier to set this table apart from other tables in the program (`1` in the example).

Adding Message Definitions

The ***message definition*** is the entry in the message table that describes the message. For example, in the previous standard format file, the entry:

```
0,"345","007",
"The system cannot recognize the date that you \n\
entered.  Please enter the date again.",
```

is a message definition. The message definition has the following parts.

0

The first number should be the index number for the help for the message. In the example, the number 0 indicates that help is not available for the message. A positive number in this position is the index number for help. A negative number in this position is the index number for help contained in the common help file. This number can only be used by the dialog manager.

program identifier

The second field (345 in the example) is a 3-character field that identifies the program. It must not conflict with identifiers already used by the system programs. Figure 6-2 on page 6-6 shows the identifiers that the system programs use. The identifier is enclosed in quotes.

message number

The third field (007 in the example) is a 3-digit field that indicates the number of the message within all messages for the program identifier. This number helps to find the message in the documentation. Message services does not use it.

message text

Messages can be any length, but must include \n (new-line) characters as appropriate to keep the final output line length, including expanded variable fields (see "Using Variable Fields in Message Text" on page 6-15), to no longer than 71 characters. Message text is enclosed in quotes.

,

A comma that is not enclosed in a set of quotes marks the parts and the end of a message definition.

};

A right brace followed by a semicolon that is not enclosed in a set of quotes indicates the end of all message definitions.

Message Index

The message index is the position of the message definition within all message definitions in the table. The first definition is number 1; the second definition is number 2. The index does not depend on any number in the table file, only on the order of the message definitions in the file. Therefore, to delete a message definition, replace it with a null definition. If you remove a definition without providing a place-holding null definition (or a new definition in that position), the index to all message definitions that occur later in the file will change.

Adding Text Insert Definitions

The text insert definition is the entry in the message table that describes the fixed text strings to insert into messages in place of the symbols **@T1**, **@T2** or **@T3**. If one or more of these symbols is in the message definition, the program can select any text insert string from the message table to replace that symbol. The method for doing that is described in "Using Variable Fields in Message Text" on page 6-15.

The requirements of the text insert definition section of the message table are that it:

- Follows the message definitions section
- Begins with the statement:

```
static msg__ins ins_defs[] = {
```

- Contains a series of text strings that are:
 - Enclosed in quotes
 - Separated by commas
- Ends with a }; (right brace and semicolon).

The example message table in Figure 6-3 on page 6-9 shows an example of the text insert definition section. That example contains the following text strings:

```
"month",
"day",
"year",
};
```

Using Messages in a Program

To use the operating system message services to display messages from a message table, the program should:

1. Include the correct header files in the program
2. Pass needed values to message services by setting system external variables (see "Using Variable Fields in Message Text" on page 6-15)
3. Use the message services routines to generate the messages.

Including Header Files

Message services uses the header files shown in Figure 6-4. Include these files in the program when using message services. These header files are in the directory **/usr/include**. Display these files (using the **pg** command) to see the exact content.

File Name	Function
msg00.h	Contains **#include** statements for the following main header files to make including those files easier: • msg01.h • msg02.h • msg03.h • msg04.h • msg05.h • msg06.h • msg08.h
msg01.h	Defines the bits of the **flags** argument to message routines.
msg02.h	Defines severity codes displayed in messages.
msg03.h	Defines origin codes that indicate where the error was detected.
msg04.h	Defines error return codes.

Figure 6-4 (Part 1 of 2). Header Files

File Name	Function
msg05.h	Defines a structure for the **msgrtrv** function when a zero value is specified for the `nbyte` argument.
msg06.h	Defines the external variables that pass values to message services.
msg08.h	Defines structures to be used when compiling a message table.

Figure 6-4 (Part 2 of 2). Header Files

Using Routines to Display Messages

After including the needed header files and setting any needed values in the message services external variables, use one of the following routines to display the message. The message can be either immediate or queued.

Generating an Immediate Message

The **msgimed** routine performs the following functions:

1. Gets message text from the message table
2. Expands standard symbols in the message text
3. Outputs the message to either **stderr** or to a specified file.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

Generating a Queued Message

The **msgqued** routine performs the following functions:

- Gets message text from the message table
- Expands standard symbols in the message text
- Outputs the message to the queued message file **/qmsg**.

Note: Queued messages are sent to **/qmsg** and generate a beep tone at the system console. They are not directed at the person using the program.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

Using Variable Fields in Message Text

A variable field is a standard symbol in the text of the message that is replaced with a value when message services displays the message. The value can be either a static (unchanging) text string or a new value each time, depending on the type of variable field used. The program passes values to message services by setting external variables. Use the following standard symbols in message definitions:

Symbol	Definition
@I1	Message services replaces this symbol with the character form of the integer value in the external variable **msgvi1**.
@I2	Message services replaces this symbol with the character form of the integer value in the external variable **msgvi2**.
@L1	Message services replaces this symbol with the character form of the long integer value in the external variable **msgvl1**.
@L2	Message services replaces this symbol with the character form of the long integer value in the external variable **msgvl2**.
@C1	Message services replaces this symbol with the null-terminated character string pointed to by the external variable ***msgvc1**.
@C2	Message services replaces this symbol with the null-terminated character string pointed to by the external variable ***msgvc2**.
@C3	Message services replaces this symbol with the null-terminated character string pointed to by the external variable ***msgvc3**.
@T1	Message services replaces this symbol with the text insert (from the text insert definitions section of the message table) selected by using the external variable **msgvt1** as an index value.
@T2	Message services replaces this symbol with the text insert (from the text insert definitions section of the message table) selected by using the external variable **msgvt2** as an index value.

Figure 6-5 (Part 1 of 2). Standard Symbols

Symbol	Definition
@T3	Message services replaces this symbol with the text insert (from the text insert definitions section of the message table) selected by using the external variable **msgvt3** as an index value.

Figure 6-5 (Part 2 of 2). Standard Symbols

Example of the Integer Symbol

If a message definition contains the message:

```
"Your value of @I1 is out of range."
```

To display the value that the operator entered, assign that value to the external variable **msgvi1** in the program before calling for the message to be displayed.

Figure 6-6 on page 6-17 shows an outline of a program to display that message. In this program, if index 3 of `xyztab1` contains the message definition of the previous example, and the value of `val` is 14, message services displays the following message:

```
Your value of 14 is out of range.
```

```
/* Define integer to hold operator input */
integer val;
/* Define integer for return code */
integer i;
/* Include required files to use message services */
#include <msg00.h>

{
   extern msg__table xyztab1;

/*
** Code that handles input and determines
** that the value is out of range.
*/

/*
** Assign bad value to external variable, and
** call the routine to display an immediate message
** to standard error, using message index 3
** and message table, xyztab1.
*/
msgvi1 = val;
i = msgimed(MSGFLTAB,&xyztab1,3);
/*
** Rest of program
*/
}
```

Figure 6-6. Example of Integer Symbol Programming

Example of the Long Integer Symbol

Inserting a long integer value into a message is the same as inserting an integer value, except it uses the external variable **msgvl1** to assign a value to the symbol **@L1**, and the variable assigned to the **msgvl1** must be of type **long**. With those exceptions, use the example for integer values as a framework to use a long integer value.

Example of the Character String Symbol

If the message definition contains the message:

```
"The day that you entered, @C1, is not\n
 a correct day of the week.  Please try again."
```

To display the string that the operator entered, assign a pointer to that string to the external variable **msgvc1** in the program. Then call for the message to be displayed. The string must end with a `\0` (null character).

Figure 6-7 on page 6-19 shows the outline of a program to display that message. In this program, if index 4 of `xyztabl` contains the message definition of the previous example and the string entered is `munday`, message services displays the following message:

```
The day that you entered, munday, is not
a correct day of the week.  Please try again.
```

```
/* Define pointer to operator input. */
char *inp;
/* define integer for return code. */
integer i;
/* Include required files to use message services. */
#include <msg00.h>

{
   extern msg__table xyztab1;

/*
** Code that handles input and determines
** that the string entered is not correct.
** Code sets inp to point to the null
** terminated string received from the user.
*/

/*
** Assign pointer inp to external variable, and
** call the routine to display an immediate message
** to standard error, using message index 4
** and message table, xyztab1.
*/
 msgvc1 = inp;
 i = msgimed(MSGFLTAB,&xyztab1,4);

/*
 rest of program
*/
}
```

Figure 6-7. Example of Character String Symbol Programming

Example of Text Insert Symbol

If the message definition contains the message:

```
"The @T1 that you entered is not\n
 a correct @T1.  Please try again."
```

Use this message to indicate errors in many areas of the program. First create a ***text insert definition*** section in the message table. See "Adding Text Insert Definitions" on page 6-11 for the format of a text insert definition section. If the text insert definition section contains the following few entries at the beginning of the table:

```
"day"           /* index 1 */
"week"          /* index 2 */
"month"         /* index 3 */
"year"          /* index 4 */
```

you can insert any of the strings `day`, `week`, `month` or `year` in place of the symbol **@T1** in the message by setting the external variable **msgvt1** to the index value (1, 2, 3 or 4) that selects the string to insert. When displaying the message, message services looks up the text string in the message table and inserts it in place of the symbol **@T1**.

Figure 6-8 on page 6-21 shows the outline of a program that displays this message. In this example, if index 3 of `xyztabl` contains the message definition of the previous example and the value of `i_text` is 3, message services displays the following message:

```
The month that you entered is not
a correct month.  Please try again.
```

```
/* Define integer for return code. */
integer i;
/* Define integer to select insert text. */
integer i_text;
/* Include required files to use message services. */
#include <msg00.h>

{
   extern msg__table xyztab1;

/*
** Code that handles input and determines
** that a field entered is not correct.
** Code sets i_text to the index value to
** select the name of the field in error.
*/

/*
** Assign i_text to external variable, and
** call the routine to display an immediate message
** to standard error, using message index 3
** and message table, xyztab1.
*/
 msgvt1 = i_text;
 i = msgimed(MSGFLTAB,&xyztab1,3);

/*
 rest of program
*/
}
```

Figure 6-8. Example of Text Insert Symbol Programming

Help

Help is text that explains difficult concepts, quick procedure steps, or other information to make it easier to use the program. You determine when to display the help from the program, and what information to include in the help. Message services provides a way to incorporate help into the program while lowering memory usage.

You can use help in the program with all of the variable symbols used with messages, including text inserts. Define text inserts in the same file that defines the help text.

Because help usually contains a lot of text, help is not compiled into the program like messages are. Instead, help is in a specially formatted help file that is not kept in memory unless it is being used. Overlaying the help file in this manner helps lower memory requirements for the program.

Note: The method for storing and displaying help described in this part of the book can also be used for messages that are not kept in memory with the program. You can put both help definitions and message definitions in the same file.

Message services provides a routine to help display help from the program. The routine is in the library file **/usr/lib/librts.a**. When using message services, link this library file with the program. In addition, two programs **gettext** and **puttext** help to change or create information in the help files. The services include:

- A routine that displays help from the program
- Header files to simplify declarations needed to use message services
- Variable field symbols in the help that message services replaces with values that you specify when it displays the help.

Help Format

Choose any format for the help text that fits the program. You can use the format described for messages (see "Message Format" on page 6-4), you can change that format, or you can define a different format. However, follow the format defined for the help file.

File Path Name

If you do not change the default path, the file that contains the help text must have the path name of:

`/usr/lib/msg/`*`program`*`_EN.m`

where the file name parts are:

program
: A program identifier that is unique.

`_EN.m`
: The ending sequence that message services requires of all help (and message) files that are not linked with the program.

Changing the File Path Name

You can specify an alternative directory to contain the help and message file for the program. Use the rules for naming the help file as described in "File Path Name." To specify the alternative directory, assign a pointer to the new path name prefix to the external variable **msgpath** in your program. For example:

```
msgpath = "/u/myprog/";
```

This statement tells message services to look in directory **/u/myprog** for the new help file. Message services looks in the default directory only if it cannot find the file in the directory you specify with the **msgpath** variable. If **msgpath** contains a null value, message services looks only in the default directory. To specify the current directory as the directory that contains the help file, set the **msgpath** variable to a pointer to a null string.

The **msgpath** variable is declared in **msg06.h**.

Changing the File Path Name for Debugging

You can also specify an alternative path name to use for the help file when testing help definitions. An alternative path name allows you to test new or changed help definitions without affecting the existing help file that is installed on the system. You can also use the alternative path name to test a new help file before installing it on the system. Use the rules for naming the help file as described in "File Path Name" on page 6-23.

To specify the alternative directory, assign the new path name prefix to the environment variable **MSGPATH** from the command line. For example:

```
MSGPATH=/u/mytest/
```

assigns the path name prefix `/u/mytest/` to the **MSGPATH** variable. Then, export the variable with the **export** command on the command line:

```
export MSGPATH
```

This operaton tells message services to look in directory **/u/mytest** for the help file. Message services looks in the default directory only if it cannot find the file in the directory specified with the **MSGPATH** variable.

Building a Help File

Message services can help to build a file of help text that is separate from the source code of the program. To build the help file, first get a file containing the standard help format from the file system. Then add help definitions to that file, and, using a program from message services, format that file for use as a help file. This method puts the help text in a file that message services can read.

Having a separate help file makes it easier to change the help text, add help text, and translate the text to another language.

Perform the following steps to build a help file and format it for use by message services. Descriptions of the commands **gettext** and **puttext** are in *AIX Operating System Commands Reference*. Refer to the following paragraphs for additional explanation for some of the steps.

1. Use the **gettext** command to create a file containing the input format for a help file. This file contains formats to fill in for including messages and text inserts, as well as help text, in the help file. For example:

   ```
   gettext myhelps
   ```

 creates a file `myhelps` that contains the framework for a help file.

2. Add help text to the proper places in the file.

3. Use the **puttext** command to format the help file. For example:

   ```
   puttext myhelps myprog_EN.m
   ```

 uses the help text in the file `myhelps` to create the help file `myprog_EN.m`. See "File Path Name" on page 6-23 for information about naming the help file.

Content of the Help Text File

When the **gettext** command gets a help file format to fill in, the file that the command creates contains entries like those in Figure 6-9 on page 6-26.

```
COMPONENT ID = prgxxx

**(((Start message)))************************************
INDEX#:
COMPSRC:
MSGSRC:
DCOMPID:
DMSGID:
STATUS:
HELP#:
TEXT:

**(((Start insert)))*************************************
INDEX#:
COMPSRC:
MSGSRC:
DCOMPID:
DMSGID:
STATUS:
TEXT:

**(((Start help)))**************************************
INDEX#:
COMPSRC:
MSGSRC:
DCOMPID:
DMSGID:
STATUS:
TITLE:
TEXT:
```

Figure 6-9. Content of Help Text Format File

The fields shown in the figure have the following meaning:

COMPONENT ID = prgxxx

Replace `prgxxx` with a 6-character identifier to indicate which program uses this help file. Do not use the characters: *** ? []** and blank.

****(((Start** *typename***)))*********************

This line is a delimiter that starts each new message and must be in the format indicated. Replace *typename* with either `message`, `insert` or `help`, as appropriate.

INDEX#:

A 3-digit field that indicates the number of the message within each type (message, insert or help). Numbers start with 001 at the beginning of the definitions for each type. Message services uses this index number to locate the text to be displayed.

COMPSRC:

Component source - Defines where the text definition for this message is located. Enter `======` for *in this file.*

MSGSRC:

Message source - A 3-digit field that defines the index number to use to display text for this message. Enter either `===` to indicate the current **INDEX#**, or a number to get text from a different message in this file.

DCOMPID:

Component ID - A 3-character field that identifies the program. It must not conflict with identifiers already used by the system programs.

DMSGID:

Message ID - A 3-digit field that indicates the number of the message within all messages for the program identifier. This number helps to find the message in the documentation. Message services does not use it.

STATUS:

Status of the message - Enter `null` to indicate the message is not used. Enter `current` to indicate the message is an active message.

HELP#:

A 3-digit index number to locate the help text for a message. This field is not required. Access this field using the **msgrtrv** call.

TITLE:

A character field of up to 79 characters that can provide a title for help text displayed on the screen. You can access this field only with the **msgrtrv** call, not with the **msghelp** call.

TEXT:

Enter the text for the message, insert or help, formatted as it appears on the screen.

Help text, like immediate messages, can be as long as needed to cover the information. Do not include any message ID for help text unless it refers to the book supplied with the program. You can also use variable fields in the help text. Refer to "Using Variable Fields in Message Text" on page 6-15 for information about using variable fields.

Using Help in a Program

To use the operating system message services to display help from the help file, the program must:

1. Include the correct header files in the program
2. Pass needed values to the message services by setting system external variables (see "Using Variable Fields in Message Text" on page 6-15)
3. Use the message services routines to display the help text.

Including Header Files

The operating system message services uses header files. Include these files in the program when using message services. These header files are described in "Including Header Files" on page 6-13.

Using Routines to Display Help

After including the needed header files and setting any needed values in the message services external variables, use the following routines to generate the help.

Displaying a Help

The **msghelp** routine performs the following functions:

1. Gets help text from the help file
2. Expands standard symbols in the help text
3. Outputs the help to either **stderr** or to a specified file.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

Putting Help in a Buffer

The **msgrtrv** routine performs the following functions:

1. Gets help text from the help file
2. Expands standard symbols in the help text
3. Outputs the help to a specified buffer.

Refer to *AIX Operating System Technical Reference* for information about the format and syntax of this routine.

Chapter 7. Monitoring Program Activities

CONTENTS

About This Chapter 7-2
Overview 7-3
Using the Trace Facilities 7-4
 Altering the Trace Configuration Files 7-7
 Using the Trace Commands 7-8
 Using the Trace Subroutines 7-10
 Creating Trace Templates 7-16
Using the Error Log Facilities 7-23
 Altering the Error Log Configuration File 7-26
 Using the Error Log Commands 7-27
 Using the Error Log Subroutines 7-29
 Creating Error Templates 7-35
Using the Dump Facilities 7-42
 Using the Dump Commands 7-45
 Example of the Dump Formatter Output 7-45

About This Chapter

This chapter contains a short overview of the three categories of monitoring software plus one longer section for each group.

The first section describes the trace facilities. The trace facilities allow you to log and format trace data. This includes a functional definition of the trace components and information on using the trace commands and subroutines.

The second section describes the error log facilities. These are similar to the trace facilities except that you log error data rather than trace data.

The third section describes the dump facilities. These are used to analyze data stored that was stored in memory at the time of a system failure.

This chapter does not contain complete usage information for the commands and subroutines used with these facilities. For more detailed information, see:

- *AIX Operating System Technical Reference*:

 errsave
 errunix
 trace_on
 trcunix
 trsave
 trc_start
 trc_stop

- *AIX Operating System Commands Reference*:

 dumpfmt
 errdead
 errdemon
 errpt
 errstop
 errupdate
 trace
 trcrpt
 trcstop
 trcupdate

- *Virtual Resource Manager Technical Reference*:

 _dmptbl
 _errvrm
 _trcvrm
 _trcgen

Overview

The system provides several software components that enable you to monitor program activities. These components can be grouped into three basic categories:

- Trace

 You use the trace facilities to monitor system performance or to aid in debugging programs. The system programs have several event classes, each of which can be turned on or off depending on your needs. For each event class, several trace points have been defined in various system software components. When a component containing a trace point is processed, the trace point generates a trace entry. You can also generate trace entries from your own programs using the three subroutines provided with the system. These subroutines allow you to generate trace entries from applications, kernel components, or the Virtual Resource Manager (VRM) component.

 All trace entries are stored in a trace log file. The trace log can be formatted into a readable trace report and sent to the display screen, a printer, or another file.

- Error Logging

 Error logging is automatically enabled when you initialize the system. It can be disabled if necessary, but it usually runs as a background process, collecting error entries generated by software components. The error entries are stored in an error log that can be formatted into a readable error report. As with the trace facilities, you can use special subroutines to generate error entries from your own programs.

- Dump

 This chapter is concerned with the VRM component dump function only.

 The system supports a VRM dump function and an AIX Operating System command called **dump**. This chapter is concerned with the VRM dump function only.

 You use VRM dump to collect data stored in memory at the time of a system failure. The dump data is written to a blank, formatted diskette. The dump program collects a pre-defined set of VRM data plus any additional data structures that you identify through the use of a VRM subroutine. The amount of dump data is limited to the capacity of a formatted high-capacity diskette.

 Once the dump data is placed on a diskette, it can be formatted and analyzed. Because the data is obtained from VRM components, you will probably not be able to interpret the output. This data is mainly used by IBM to determine the cause of a system failure.

Using the Trace Facilities

The trace facilities are used to monitor changes to variable data within software components. Tracing can be used as a debugging aid and to check the performance of various sections of code. Some of the basic trace terms are defined below:

trace entry A data structure containing a header of identifying information plus up to 20 bytes of defined data. Trace entries are generated by trace points and written to a trace log file that can be formatted by the trace formatter.

trace point A group of code statements that generates a trace entry from within a software component. Trace points are assigned to an event class which can be active or inactive. Trace points with active event classes are able to generate trace entries.

event class A number assigned to a group of trace points that relate to a specific subject or system component. The defined event classes are listed in the trace profile, **/etc/trcprofile**.

hook ID A unique number assigned to a specific trace point. All trace entries include the hook ID of the originating trace point in the trace entry header. Pre-defined trace points use assigned hook IDs ranging from 0 to 299. User-defined trace points can choose hook IDs ranging from 300 to 399.

How you use the trace facilities depends on what you want to accomplish:

- To choose the event classes that you want to trace, you should learn how to alter the default trace profile, **/etc/trcprofile**, and how to create your own trace profile.
- To start a trace session, you should use the **trace** command.
- To end a trace session, you should use the **trcstop** command.
- To format a trace log file, you should learn how to use the **trcrpt** command.
- To change the name or size of the default trace log file, you should learn how to alter the **/etc/rasconf** file.
- To create trace points that generate trace entries, you should learn how to use the trace subroutines: **trcunix**, **trsave**, **_trcvrm**, and **_trcgen**.
- If you create your own trace entries, you should learn how to create a trace template for each type of trace entry. You also need to learn how to use the **trcupdate** command to add trace templates to the template file, **/etc/trcfmt**.

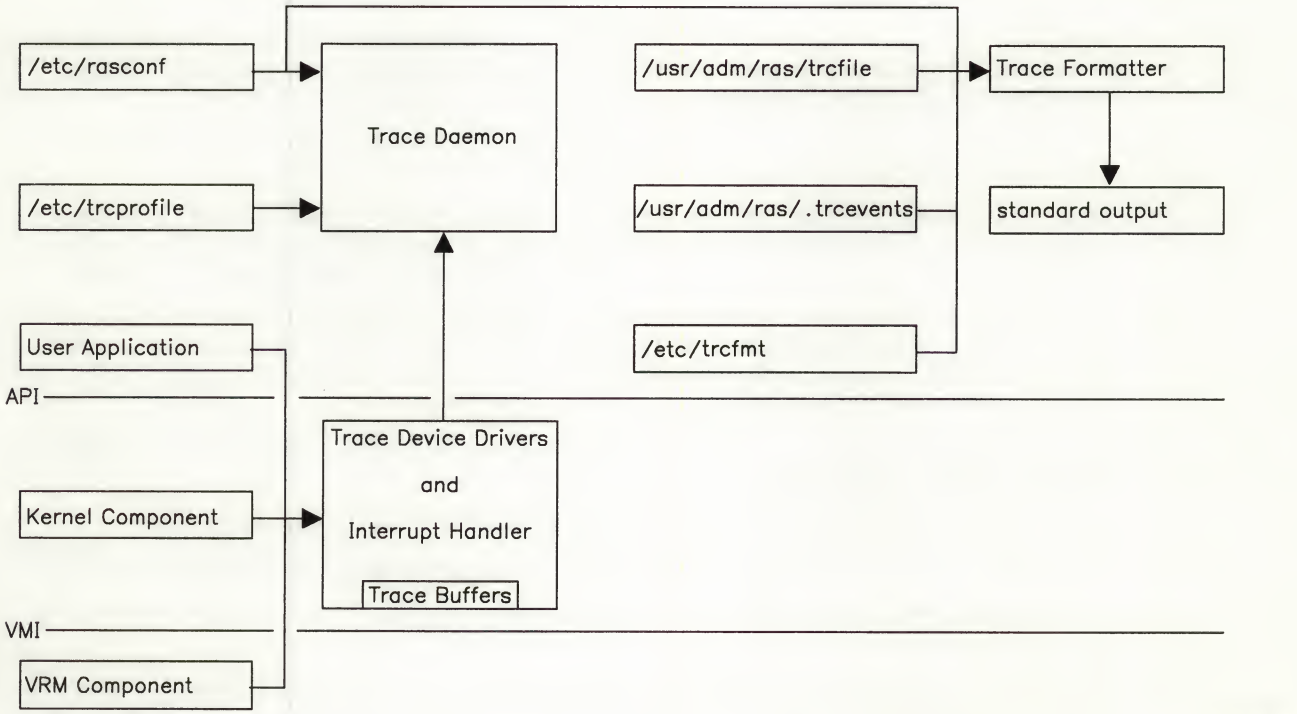


Figure 7-1. Trace Components

Figure 7-1 shows how data is passed between the various files and components that constitute the trace facilities. The lines labeled API, which stands for Application Program Interface, and VMI, which stands for Virtual Machine Interface, show the logical distinction between application programs, kernel components, and the VRM component. The lines connecting the files and components show where data comes from and where it goes.

The components and files in Figure 7-1 are described on the next page. They are explained in more detail elsewhere in this chapter in the appropriate sections.

The following descriptions start at the point where the trace entries are first generated and end where the trace entries are formatted and sent to standard output.

- Applications, kernel components, and VRM components generate trace entries using trace points placed at strategic places in the execution path. The system programs contain several pre-defined trace points relating to various event classes.
- The trace device drivers collect the trace entries in trace buffers. There is one trace buffer for each of the three trace subroutines.
- The **/etc/rasconf** file contains configuration data. For the trace daemon, it defines the name and size of the trace log file to be opened to receive trace entries. For the trace formatter, it defines the default trace log file if none is specified when it is invoked.
- The default trace profile is **/etc/trcprofile**. This file contains a list of the defined event classes. An event class is either active or inactive. If an event class is active, the trace points related to that event class will generate trace entries. You can use the default trace profile, or create your own trace profile.
- The trace daemon is an important part of the trace facilities. When it is initialized, it performs three major tasks:
 - It reads the trace profile to determine which event classes should be active.
 - It opens the file specified in the trace stanza in **/etc/rasconf** as the trace log file.
 - It begins reading the trace buffers as they become full and writes them out to the trace log file.
- The default trace log file is **/usr/adm/ras/trcfile**. This file stores all of the trace entries generated by software programs. If the trace log file becomes full, the newest trace entry overwrites the oldest trace entry.
- The **/usr/adm/ras/.trcevents** file contains lists of event classes and the hook IDs associated with those event classes. The ***hook ID*** is a specific number associated with a particular trace point. The trace formatter uses information in this file to count the trace entries that occur for each event class. You do not edit this file directly. It is automatically updated when you use the **trcupdate** command.
- The trace format file, **/etc/trcfmt**, contains trace templates that determine how each trace entry appears when it is formatted. The pre-defined trace entries also have pre-defined templates. If you generate trace entries from your own programs, you need to define trace templates for those entries.
- The trace formatter formats the trace entries in a trace log file into a readable format. If a trace log file is not specified when the trace formatter is invoked, it uses the file specified in **/etc/rasconf**. The formatted trace entries are sent to standard output and can therefore be sent to the display screen, a file, or a printer.

Altering the Trace Configuration Files

There are two basic files that you can alter to change the operation of the trace facilities. The first is the ***trace profile***, which is used by the trace daemon to set up three bit masks that show which event classes are active. The trace daemon reads the trace profile when it is initialized. A trace profile contains a single line entry for each defined event class. Each line entry begins with an * (asterisk). Only those event classes that have had the * removed will be able to generate trace entries during a trace session.

You edit a trace profile using a standard text editor. You can create several trace profiles, each designed for different tracing needs. When you run **trace**, you can specify which trace profile it should use. If you do not specify a trace profile, it uses the default trace profile, **/etc/trcprofile**. To create your own trace profile, copy the default trace profile into your current directory and edit the new copy.

If you have defined trace points in your own programs, you need to have the trace daemon activate the **User-defined Events** event class. It appears in the trace profile as:

```
*      150 User-defined Events
```

To have the trace daemon activate this event class, remove the * from the beginning of that line in the trace profile. It should now appear as:

```
       150 User-defined Events
```

If you do not enable this event class, your trace points will not be able to generate trace entries in the trace log file.

The other file that you can alter is the **/etc/rasconf** file. This is a configuration file that is read by the trace daemon and the trace formatter. It contains various types of information, but the information you are interested in is the stanza that defines the name and maximum block size of the default trace log file. It appears in **/etc/rasconf** as:

```
/dev/trace:
    file = /usr/adm/ras/trcfile
    size = 80
```

Other users may change the contents of this file. You may want to change the trace log file name if the current trace log is full, or if you want to keep different logs for different trace sessions.

Each time **trace** is invoked, the trace daemon checks **/etc/rasconf** to see which file it should open to receive trace entries. All trace entries generated during the trace session are directed to that file. When you use the trace formatter, it uses the file name in **/etc/rasconf** as the default trace log file if a trace log file is not specified. If you have several trace log files, you can format up to 10 at a time by invoking the trace formatter with a list of file names.

Using the Trace Commands

There are four commands associated with the trace facilities. They each perform one basic task:

trace Starts a trace session. This command accepts a trace profile as an input file name parameter. Once a trace session is initiated, all trace points with active event classes will generate trace entries if their particular component is run.

If you do not specify a trace profile, the default trace profile, **/etc/trcprofile**, is used. The following example invokes **trace** with a user-defined trace profile. Note that an & (ampersand) is required at the end of the command line. This causes **trace** to run as a background process:

```
trace /u/myfile/myprof &
```

trcstop Stops a trace session. Any trace entries remaining in the trace buffers are written out to the trace log file. Any open files are closed and the trace daemon is terminated. This command does not require any input parameters.

trcrpt Formats trace entries contained in trace log files. This command accepts one to ten trice log file names as input parameters. If you do not specify a trace log file, it uses the default trace log file specified in **/etc/rasconf**. The formatted trace entries are sent to standard output.

The following example formats two trace log files named **trace1** and **trace2** and sends the output to the printer:

```
trcrpt trace1 trace2 | print
```

Because all trace entries are time-stamped, the trace formatter can also format a subset of a trace log file consisting of trace entries that fall within a certain time interval. You specify a starting time and end time using the **-s** and **-e** flags. The time is specified as *MMddhhmmyy* (month, day, hour, minute, year).

The following example formats the default trace log entries ranging from January 3, 1985 at 11 a.m. to 11:06 a.m. of the same day:

```
trcrpt -s0103110085 -e0103110685
```

trcupdate Adds, updates, or deletes trace templates in the **/etc/trcfmt** file. Also updates the **/usr/adm/ras/.trcevents** file. Before using this command to add or update trace templates, you need to create an input file with the extension `.trc`. This file will contain two types of entries:

- Template definitions. These contain a + (plus sign) in the first column to specify that the template is to be merged into the master template file. A single template definition may require several lines. The + is only required on the first line of each template.
- Template deletions. These contain a - (minus sign) in the first column, one blank, and the hook ID of the template you want to delete.

 Note: Do not delete the pre-defined trace templates (hook IDs 0 to 299). These are required to format the pre-defined trace points imbedded in the system programs.

The following example shows a file that will add trace template 330 to the master template file and delete templates 320 and 321. Note that the first line in the file must be entered as shown. It is used by the **trcupdate** command to verify that this file contains template information.

```
* /etc/trcfmt
+ 330 1.0 InitPtr Printer PtrNode A8: \n: \t: \
          PtrType D4: PtrActv B0.1, 1 Yes, 0 No:
- 320
- 321
```

To update a trace template, you need to create a new template with the same hook ID but with a version and release number (*VV.RR* field) greater than the version and release number of the trace template that you want to update. When **trcupdate** merges the `.trc` file into the master template file, it will replace the old template with the updated one.

The trace formatter writes the total number of trace entries attributed to each event class at the end of a trace report. If you want trace entries generated by new trace points to be counted under the **User-defined Events** event class, you must also create a file with an extension of `.evt`.

The `.evt` file contains entries that specify an event class number and any new hook IDs that you want to associate with that event class. It is merged into the **/usr/adm/ras/.trcevents** file when you use the **trcupdate** command. For user-defined trace entries, the event number is always 150. Thus, to add hook ID 330 to the **User-defined Events** event class, you would create a file as shown below. Note that the first line in the file must be entered as shown:

```
* /ras/.trcevents
150 330
```

Once you create a file containing new or updated template definitions and/or template deletions, you can use **trcupdate** to process it. In the following example, the template definitions and deletions are contained in

a file called **newtemps.trc**. The event class and hook ID lists are contained in a file called **newtemps.evt**:

```
trcupdate  newtemps
```

Using the Trace Subroutines

Before using the trace subroutines, you need to understand how the trace daemon uses the trace profile and how event classes are represented as active or inactive in the system.

After the trace daemon reads the trace profile, it sets up three tables in memory. Each table consists of a one-word array of flag bits. Each bit is called a ***channel*** and is named after the bit position it occupies. The most significant bit is called **channel 0** and the least significant bit is called **channel 31**. Each event class is represented by one or more of these channels. When the trace daemon reads the trace profile and sees that an event class should be active, it sets the appropriate channels in each channel table.

When the trace daemon has read the entire trace profile, it sends one channel table to each of the three trace device drivers. When one of these drivers is called by a trace subroutine, the device driver compares the channel number, which is a required part of the trace entry, against its own channel table. If the channel is active, it puts the trace entry in a trace buffer. If not, it does nothing; however, in each case it returns a successful return code unless there is a problem in the calling procedure.

With 32 bits available for each channel table, it is possible to have 96 different event classes. However, an event class may use the same channel in more than one table. Thus, the actual number of event classes is less than 96. The **User-Defined Events** event class uses channel 31 in each of the three channel tables. This allows you to create user-defined trace entries using any of the three trace subroutines. User applications can use the **trace_on** subroutine to see if a specific channel is enabled.

Note: If you look at the trace profile, you will see a number placed before each event class. This number identifies a specific event class; however, it is not a channel number. Channel numbers can only be in the range from 0 to 31.

When you use a trace subroutine, one of the input parameters contains a channel number and hook ID to identify which event class that trace entry belongs to and which trace point generated the call. This is called the ***trace ID***. Bits 0 through 4 of the trace ID contain the binary representation of the channel number. Bits 5 through 15 contain the binary representation of the hook ID. Thus, for user trace entries, bits 0 through 4 should always be set to 1 (channel 31) and the hook ID should be a number ranging from 300 to 399.

The parameters used by the trace subroutines are briefly explained in the following:

trcunix Generates a trace entry from an AIX Operating System application and requires two input parameters. The first is the address of a buffer containing the 2-byte trace ID and up to 20 bytes of trace data. The second is the length of the buffer.

trsave Generates a trace entry from a kernel component and requires three input parameters. The first is a 2-byte trace ID. The second is the length of the data buffer. The third is the address of the data buffer.

_trcvrm Generates a trace entry from a VRM component and requires three input parameters. The first is a 2-byte trace ID. The second is a buffer of trace data and the third is the length of the data buffer.

_trcgen Generates a generic trace entry from a VRM component and requires four input parameters. The first is the address of the generic trace buffer. The second is a 2-byte trace ID. The third is a structure containing a 2-byte IODN, a 2-byte IOCN, 4 bytes of user header data, and 4 bytes containing the actual number of bytes of the trace data buffer. The fourth parameter is the address of a buffer of trace data.

trc_start Starts the trace daemon from a process and requires the following parameters:

outpath Specifies the file name that the trace daemon uses to store the trace data. This file name must be different from the default file name in **/etc/rasconf**.

entsize Specifies the size in bytes for the entries that the application logs. If the value is 0, the default size is used.

numents Specifies the number of entries the kernel buffer will hold. If **numents** is not used, the buffer size is taken from **/etc/rasconf**. If the value of **numents** is 0, or if an invalid (non-numeric) argument is supplied, the trace daemon calculates the buffer size using the entry size and a default of 1024 entries.

last_only Specifies whether the trace deamon should log data continuously or only save the last buffer filled before the daemon stops. A value of 0 indicates that the trace daemon should log data continuously. A value of 1 indicates that the daemon should only save the last buffer filled before the daemon stops.

structure t_struct
Contains the following control values returned by the daemon:

- Process ID of the daemon

- Address of the trace buffer
- Length of the trace buffer
- Channel mask used by the process when logging trace data. The channel mask is the decimal number of the bit position assigned to the trace channel, shifted left 11 bits.

trc_stop Stops the trace daemon from the process ID started by **trc_start**.

When you create your own trace points, remember that each trace point should use a unique hook ID in the range from 300 to 399. You must also create trace templates for your trace points. Here are some general guidelines:

- Put a trace point at the beginning or end of an important function. The trace data should show the values of any important data structures or I/O parameters.
- If a function takes a significant amount of processing time, you may want to put a trace point at its entry and exit points. This will allow you to trace how long it takes to process different types of input parameters.
- Put a trace point at a critical junction in the logic flow of a component.
- Do not put a trace point inside a loop that is repeated many times if you can catch the important data before the loop begins.
- Try to limit the trace entry data to four variables or less.

The trace device drivers have access to the channel tables, so they know whether or not to put a trace entry in the trace buffers. However, you may want to explicitly check to see if your channel is active before you call a trace subroutine so you can design the trace point to skip around data collection code if the channel is not on. For kernel components, the usual procedure is to perform the subroutine call without checking for an active channel because kernel trace points do not contain much data.

A sample trace point for application level components is shown in Figure 7-2 on page 7-13. Notice how it uses the **trace_on** subroutine to see if the **User-Defined Events** channel (31) is on before calling the **trcunix** subroutine. Also, notice how the trace ID is created. First, you shift the variable containing the channel number left 11 bits so that it is in bits 0 through 4. Then you perform an OR operation on that variable with the variable containing the hook ID. The result is a trace ID containing the channel number in bits 1 through 4 and the hook ID in bits 5 through 15.

```
#include <sys/trace.h>
#include <trcdefs.h>
#include <stdio.h>

main(argc, argv)
   int argc;
   char **argv;
{
   extern int trace_on();
   extern int trcunix();
   int chan_no = 31;
   long channel = 0x1;    /* Set bit 31 (user-defined channel) to 1 */
   long hookid = 300;     /* Define the hook ID for this trace point */
   int tracing;
   int rc = 0;
   struct
   { unsigned short traceid;
      char data[20];
   } trcstruct;

   /* See if channel 31 is active.  */

   if ( (tracing = trace_on(channel) ) < 0)
   {
      fprintf(stderr,"trace_on failed\n");
      return(-1);
   }
```

Figure 7-2 (Part 1 of 2). Example Program Fragment Showing Use of trcunix Subroutine.

```
   /* Imbed this trace point anywhere in your program code. */

   if (tracing)
   {
      trcstruct.traceid = ((chan_no<<11) | hookid);
      sprintf(trcstruct.data, "This is trace data.");
      if (trcunix(&trcstruct, sizeof(trcstruct)) < 0)
      {
         fprintf(stderr,"trcunix failed\n");
         rc = -1;
      }
   }
   exit(rc);
}   /* end main */
```

Figure 7-2 (Part 2 of 2). Example Program Fragment Showing Use of trcunix Subroutine.

Figure 7-3 on page 7-15 creates a trace entry from within an AIX Operating System kernel component. Notice how it uses a defined constant (TR_USER) to set the channel number in the trace ID parameter. TR_USER contains channel number 31 in bits 0 through 4. This is only available for kernel components using the **trsave** routine.

```
#include <sys/trace.h>
#define TRACEDATA 1

function()
{
   unsigned short hookid = 310;
   int t_data;

   /* The following trace point can appear anywhere in your function. */

   t_data = TRACEDATA;
   trsave((TR_USER | hookid), sizeof(int), &t_data);
}   /* end function */
```

Figure 7-3. Example Program Fragment Showing Use of trsave Subroutine.

Creating Trace Templates

The trace formatter uses ***trace templates*** to determine how the data contained in trace entries should be formatted. All trace templates are stored in the master template file, **/etc/trcfmt**. After you create some new or updated trace templates, place them in a file that can then be merged into the master template file using the **trcupdate** command. The **trcupdate** command requires that trace templates identify themselves by placing a + and a blank in the first two columns of the first line of the trace template.

Trace templates contain four required fields and zero or more data description fields. The required fields identify the template hook ID, the version and release number, the hook ID label, and the event class label. The data description fields contain formatting information for the trace entry data and can be repeated as many times as is necessary to format all of the trace data in the trace entry.

Trace entries are formatted and written to standard output one entry at a time. For each entry, the trace formatter performs the following operations:

1. Locates the trace template corresponding to the hook ID in the trace entry.
2. Writes the time the entry was generated. It also writes a sequence number that shows when the entry was generated as an interval within a second. This sequence number allows the formatter to sort trace entries that are generated during the same second. They can then be formatted and written in the order in which they occurred. These fields are placed in the trace entry header by the trace subroutine.
3. Writes the PID, IODN, and IOCN contained in the trace entry header. These fields are placed in the trace entry header by the trace subroutine.
4. Writes the hook ID and event class. You can use the actual numbers that equate to the hook ID and event class, or you can create a name up to eight characters for each field. The trace formatter will write whatever is placed in the field.
5. Writes up to 20 bytes of data according to the *data_descriptors* in the trace template, if any are present. Any data that occurs after the last *data_descriptor* in the template is ignored. A *data_descriptor* does the following:
 a. Writes the data label (*d_label*), if it is specified.
 b. Writes the actual data according to the *format* field and the optional match fields. You can define match fields that will replace data values with descriptive labels or change to a different *data_descriptor* depending on the current data value.

Figure 7-4 on page 7-17 shows the syntax used to define a trace template. Figure 7-5 on page 7-18 defines each of the fields in a trace template.

Trace template lines can be as long as you need. You can continue a trace template on another line by adding the \ (backslash) character to the end of the line where the split occurs. Note that \ is not a substitute for a required blank. A blank is also required after a colon or comma.

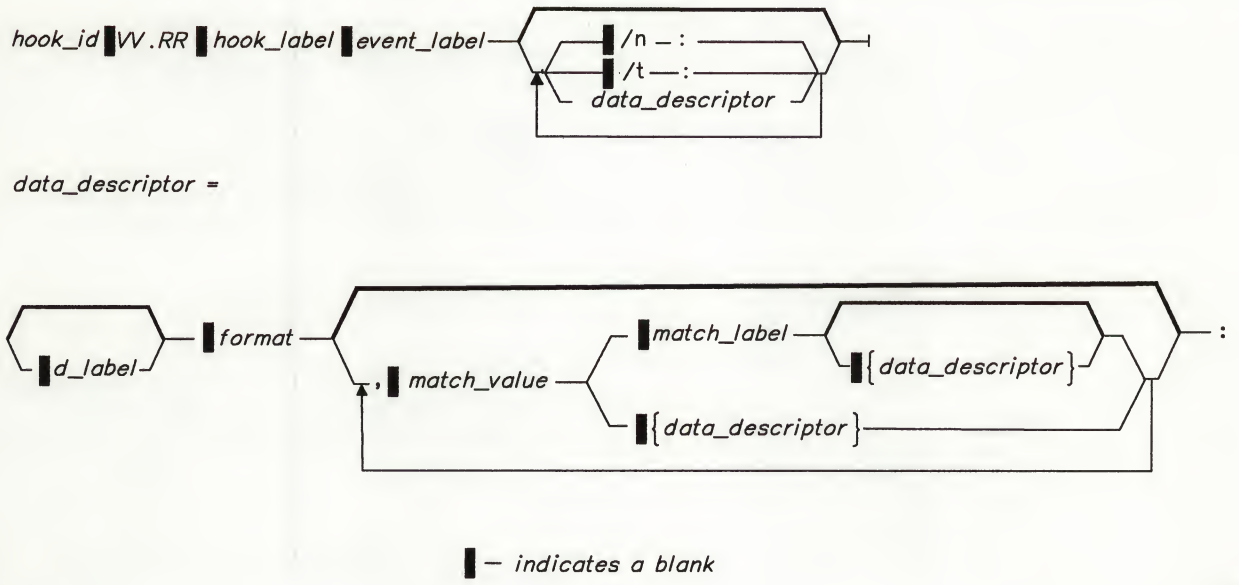


Figure 7-4. Trace Template Syntax

Field	Description
hook_id	The 3-digit hook ID of the trace entries that use this template.
VV.RR	A number representing the version and release level of this template.
hook_label	A label field of up to 8 characters that describes the hook ID.
event_label	A label field of up to 8 characters that describes the event class.
\n	A special string used to start output data on a new line.
\t	A special string used to start output data after one tab unit.
d_label	A label field of up to 8 characters that describes the trace entry data.
format	An alphanumeric code that defines the format of the trace data.

Code	Format of Data
A*m*	ASCII string of *m* characters.
B*m*	Binary string of *m* bytes.
B*m.n*	Binary string of *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
D2	Decimal short integer.
D4	Decimal long integer.
F4	Floating-point number of type *float*, rounded to four places.
F8	Floating-point number of type *double*, rounded to four places.
O*m*	Omit (do not write) the next *m* bytes.
O*m.n*	Omit (do not write) the next *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
U2	Unsigned decimal short integer.
U4	Unsigned decimal long integer.
X*m*	Hexadecimal number of *m* bytes.

match_value	A value with a data type the same as the *format* field. If you specify a *match_value*, you must also specify a *match_label* or *data_descriptor*.
match_label	A character field up to 120 characters that replaces output data that matches a *match_value* field.
data_descriptor	A field containing formatting information for a portion of the output data.

Figure 7-5. Fields in a Trace Template

Replacing Values in the Output Data

The *match_value* field provides a function that is similar to a switch statement in programming. You can use this function to do either or both of the following:

- Replace an output data value with a character string that describes the data. For example, you can replace a numeric error code with a description of that error. The replacement character string is called a *match_label.*
- Change the data descriptor used to format the output data. For example, you can design trace points that will record different data at different times, depending on the state of your machine or program. You can then define a set of *match_values* with a different *data_descriptor* for each type of data.

You can specify any number of *match_value* fields for a particular output data value as long as each field is accompanied by a *match_label* and/or a *data_descriptor* field. If the output data value matches *match_value*, the trace formatter uses the *match_label* and/or *data_descriptor* associated with the *match_value.*

You can use the special string * as a *match_value.* This will match any output data value in the current data field. When the trace formatter finds this string, it does not check any *match_values* that occur after this string. Therefore, you should use this string as the last *match_value* in a list to provide a default action if none of the other *match_values* are matched. If no match is found, the data value is written.

If a new *data_descriptor* follows the *match_value*, the trace formatter uses the formats from the new *data_descriptor* until one of the following occurs:

- It has formatted all of the trace entry data.
- It has used all of the defined *data_descriptors* for the current output data.

The trace formatter then looks for a new *data_descriptor* field to define the format for the rest of the data in the trace entry. If a new *data_descriptor* is not found, it writes the data literally according to the format given in the *format* field of the last *data_descriptor.*

Note: The trace formatter compares a *match_value* field and the current output data as two character strings. For real numbers (*F* format), it uses the **printf** subroutine to round the data to four decimal places. If you plan to use a *match_value* field with output data formatted as a real number, the *match_value* field should use the same precision, four decimal places.

Appearance of the Formatted Output Data

The following examples show you how to format data using a single *data_descriptor*. The data is defined as 8 bytes consisting of IBM4341A. The *data_descriptor* fields have the following values:

d_label	PtrNode
format	A8
match_value	IBM4341A
match_label	Model4X

Each of the following examples shows a *data_descriptor* using a combination of *data_descriptor* fields and how the output data would look if it used that *data_descriptor*:

- If you just specify a *format* field, the *data_descriptor* would be:

  ```
  A8:
  ```

 The formatted trace report would show the data as:

  ```
  IBM4341A
  ```

- If you specify *d_label* and *format* fields, but do not specify a *match_value* field, the *data_descriptor* would be:

  ```
  PtrNode A8:
  ```

 The formatted trace report would show the data as:

  ```
  PtrNode=IBM4341A
  ```

- If you specify *format*, *match_value*, and *match_label* fields, but do not specify a *d_label* field, the *data_descriptor* would be:

  ```
  A8, IBM4341A Model4X:
  ```

 The formatted trace report would show the data as:

  ```
  Model4X
  ```

- If you specify *d_label*, *format*, *match_value*, and *match_label* fields, the *data_descriptor* would be:

  ```
  PtrNode A8, IBM4341A Model4X:
  ```

 The formatted trace report would show the data as:

  ```
  Ptrnode=Model4X
  ```

Trace Template Example

This section shows you how a trace entry using a sample trace template would appear after being formatted. In the example, a trace point belonging to the **Printer** event class generated a trace entry during printer initialization. The header data for the trace entry includes the hook ID, which in this example is 330. The trace entry data is listed below:

`PtrNode` The printer node ID (8 ASCII characters). The *format* for this data would be A8.

`PtrType` The printer type (a decimal long integer). The *format* for this data would be D4.

`PtrActv` A 1-bit flag that indicates if the printer is active (1) or inactive (0). The *format* for this data would be B0.1.

Figure 7-6 shows a sample trace template for trace entries with a hook ID of 330. Notice the newline and tab descriptors before the `PtrType` field. This will cause the `PtrType` field to appear on the next line and indented one standard tab unit. Also, notice that two pairs of *match_value* and *match_label* fields are defined for the 1-bit `PtrActv` flag. This tells the trace formatter to write `Yes` if the flag equals `1` and `No` if the flag equals `0`:

```
330 1.0 InitPtr Printer PtrNode A8: \n: \t: \
        PtrType D4: PtrActv B0.1, 1 Yes, 0 No:
```

Figure 7-6. Example of a Trace Template for hook ID 330

Figure 7-7 on page 7-22 shows how a trace entry with a hook ID of 330 might appear in a formatted trace report. In the report, the example trace entry is the seventh formatted entry. Notice that the last part of the report shows a list of event classes and how many trace entries belong to each event class.

```
                              TRACE LOG REPORT

File: /usr/adm/ras/trcfile

Fri Jan  3 13:19:35 1985
System: ****            Node: ****

Version: 0                      Machine: ****

TIME         SEQ   PID  IODN  IOCN TYPE     HOOK     DATA

23:23:34.01 0001 00195             I/O_Sys  ioctl[x] fildes=5 request=23
        arg=536872136
23:23:34.01 0002 00195             I/O_Sys  ioctl[x] fildes=1 nbyte=28
23:23:34.01 0003 00195             I/O_Sys  write[x] fildes=1 nbyte=40
23:23:34.01 0004 00195             I/O_Sys  write[x] fildes=1 nbyte=31
23:23:34.01 0005 00142             I/O_Sys  write[x] fildes=2 nbyte=1
23:23:34.01 0006 00142             I/O_Sys  write[x] fildes=2 nbyte=2
23:23:34.01 0007 00140             Printer  Initptr  PtrNode=IBM4341
        Ptrtype=3 PtrActv= No
23:23:34.01 0008 00196             I/O_Sys  read[x]  fildes=3 nbyte=128
23:23:34.01 0009 00196             I/O_Sys  access   errno=2 filemode=1
23:23:34.01 0010 00196             I/O_Sys  access   errno=2 filemode=1
23:23:34.01 0011 00196             I/O_Sys  ioctl[x] fildes=0 request=17
        arg=1073737112

Summary of event counts.
Event 0: 2              Event 66: 8
Event 150: 1

Total number of events: 3
```

Figure 7-7. Example of Output from the Trace Formatter

Using the Error Log Facilities

The error log facilities are used to record errors that may occur in the system. These errors can be in hardware or software, and can be of several different types. Some of the basic error terms are defined below:

error entry A data structure containing a header of identifying information plus several bytes of defined data. Error entries are generated by error points and written to an error log file that can be formatted by the error formatter.

error point A group of code statements that generates an error entry from within a software program. Error entries are generated when a software or hardware component encounters an error.

error ID This is part of the data required by an error entry. It is a unique combination of three hexadecimal digits that identifies the component that generated the error entry.

error identifier A three-character code used to identify error templates and to specify which error entries the error formatter should process. This code is based on the error ID; however, it use alphanumeric characters instead of hexadecimal digits.

How you use the error facilities depends on what you want to accomplish:

- To start the error log facilities, you should use the **errdemon** command. You usually will not have to use this command because the default initialization process is set to turn error logging on.
- To stop error logging, you should use the **errstop** command. You must have superuser authority to use this command.
- To format the error log files, you should learn how to use the **errpt** command.
- To change the name or size of the default error log files, you should learn how to alter the **/etc/rasconf** file.
- To create error points that generate error entries, you should learn how to use the error subroutines: **errunix**, **errsave**, and **_errvrm**.
- If you create your own error entries, you should learn how to create an error template for each type of error entry. You also need to learn how to use the **errupdate** command to add error templates to the template file, **/etc/errfmt**.

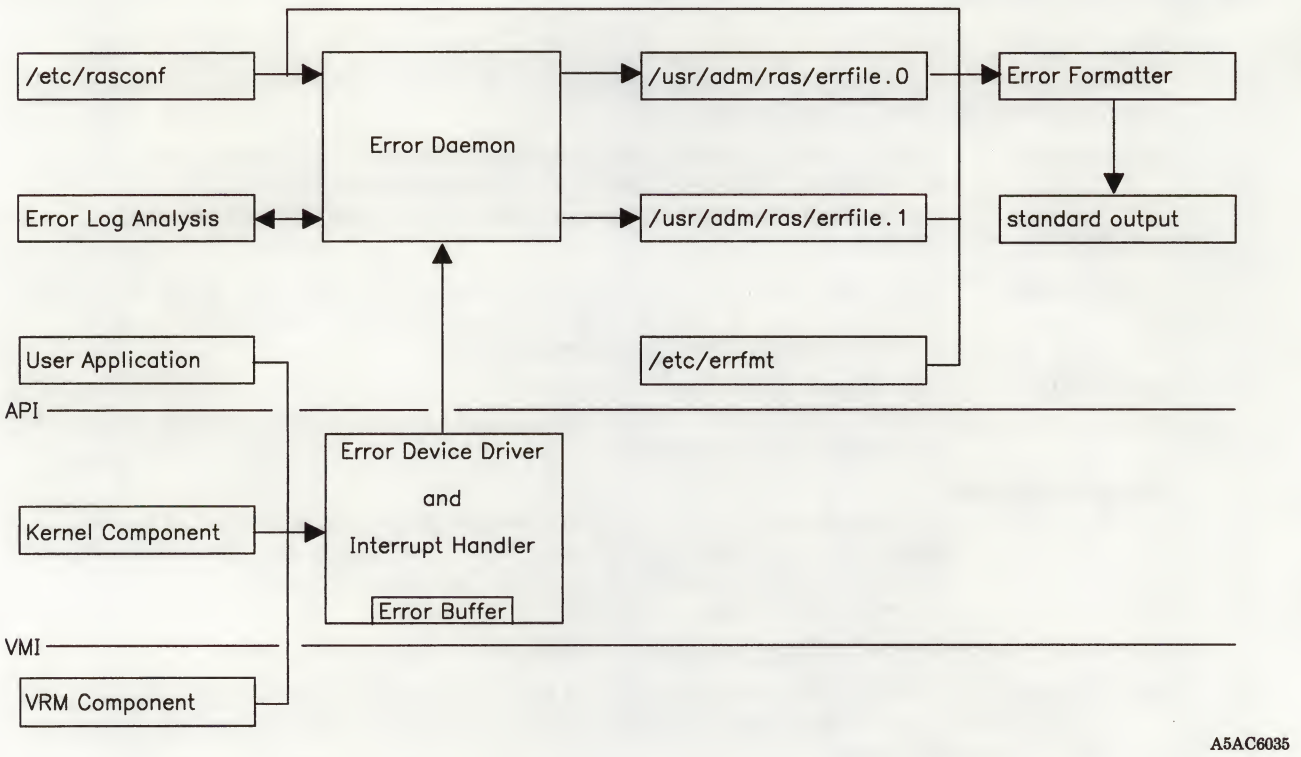


Figure 7-8. Error Components

Figure 7-8 shows how data is passed between the various files and components that constitute the error facilities. The lines labeled API, which stands for Application Program Interface, and VMI, which stands for Virtual Machine Interface, show the logical distinction between application programs, kernel components, and the VRM component. The lines connecting the files and components show where data comes from and where it goes.

The components and files in Figure 7-8 are described on the next page. They are explained in more detail elsewhere in this chapter in the appropriate sections.

The following descriptions start at the point where the error entries are first generated and end where the error entries are formatted and sent to standard output.

- Applications, kernel components, and VRM components generate error entries using error points. The system programs contain several pre-defined error points that generate error entries when necessary.
- The error device driver collects the error entries in the error buffer. There is one error buffer that collects error entries from all three error subroutines.
- The **/etc/rasconf** file contains configuration data. For the error daemon, it defines the name and size of the file that the error daemon uses to open two error log files. For the error formatter, it defines the default error log file name if a file name is not specified when it is invoked.
- The error daemon is an important part of the error facilities. When it is initialized, it performs the following major tasks:
 - It opens two error log files by appending the extensions **.0** and **.1** to the file name specified in the error stanza in **/etc/rasconf**.
 - It checks the non-volatile RAM and if there is data, generates an error entry.
- The default error log files are **/usr/adm/ras/errfile.0** and **/usr/adm/ras/errfile.1**. These files are organized as a circular buffer and store all of the error entries generated by the software programs. If one of the error log files becomes full, the next error entry is written into the other file, discarding any entries that may have been in the file.
- The error format file, **/etc/errfmt**, contains error templates that determine how each error entry appears when it is formatted. The pre-defined error entries also have pre-defined templates. If you generate error entries from your own programs, you need to define error templates for those entries.
- The error formatter formats the error entries in the error log files into a readable format. If an error log file is not specified when the error formatter is invoked, it uses the file specified in **/etc/rasconf**. It reads the error files and sends error entries to the error log problem determination program, **errpd**. The **errpd** program analyzes the error entries and returns probable cause information to the error formatter. When **errpd** is finished, the error formatter appends the **errpd** analysis information, if any, onto the error entry and formats the entry. The formatted error entries are sent to standard output and can therefore be sent to the display screen, a file, or a printer.

Altering the Error Log Configuration File

There is one file that you can modify before starting the error daemon. This is the **/etc/rasconf** file, which is a configuration file that is read by the error daemon and the error formatter. It contains various types of information, but the information you are interested in is the stanza that defines the name and maximum block size of the default error log file. This information appears in **/etc/rasconf** in the following form:

```
/dev/error:
    file = /usr/adm/ras/errfile
    size = 50
```

Another user may change the contents of this file. You may want to change the error log file name if the current error log files are full, or if you want to keep different logs for different types of error entries.

Each time **errdemon** is invoked, the error daemon checks **/etc/rasconf** to see which files it should open to receive error entries. The error daemon does not open the actual file name specified in **/etc/rasconf**. It merely uses the file name as the basis for two other file names with extensions of **.0** and **.1**. The file names with the extensions are the ones that are actually opened. For example, if the error stanza in **/etc/rasconf** is:

```
/dev/error:
    file = /usr/adm/ras/errfile
    size = 50
```

the error daemon will open two files using the names:

```
/usr/adm/ras/errfile.0
/usr/adm/ras/errfile.1
```

All error entries generated while the error daemon is active are directed to these two files. When you use the error formatter, it uses the file name in **/etc/rasconf** as the default error log file if an error log file is not specified. The error formatter adds the **.0** and **.1** extensions to the error log file name to determine the actual names of the two error log files.

Using the Error Log Commands

There are five commands associated with the error log facilities. They each perform one basic task:

errdemon Starts an error session. Once the error daemon is initiated, all error points will generate error entries whenever their error conditions are met. This command is usually placed in the command file **/etc/rc**, which is run when the system is initialized. This command does not require any input parameters.

errstop Stops the error daemon. Any error entries remaining in the error buffer are written out to the error log file. Any open files are closed and the error daemon is terminated. This command does not require any input parameters. Note that it is not necessary to stop the error daemon in order to format the error log files.

errpt Formats error entries contained in error log files. This command accepts one or more error log file names as input parameters. If you do not specify an error log file, it uses the default error log file specified in **/etc/rasconf**. The formatted error entries are sent to standard output.

The following example formats two error log files named **myerr.0** and **myerr.1** and sends the output to the printer. Notice that you use the file name **myerr** as input. The error formatter automatically appends the extensions and looks for those files, not the file named in the command:

```
errpt myerr | print
```

There are several options available to format subsets of the entire error log. See the description of the **errpt** command in *AIX Operating System Commands Reference*.

errupdate Adds, updates, or deletes error templates in the **/etc/errfmt** file. Before using this command to add or update error templates, you need to create an input file with the extension `.err`. This file will contain two types of entries:

- Template definitions. These contain a + in the first column to specify that the template is to be merged into the master template file. A single template definition may require several lines. The + is only required on the first line of each template.
- Template deletions. These contain a - (minus sign) in the first column, one blank, and the error identifier of the template you want to delete.

 Note: Do not delete the pre-defined error templates. The only templates that you should consider deleting are those that are user-defined. User-defined error templates have an error identifier beginning with **U** or **HF**.

The following example shows a file that will add error template U13 to the master template file and delete templates U11 and U12. Note that the first line in the file must be entered as shown. It is used by the **errupdate** command to verify that this file contains template information.

```
* /etc/errfmt
+ U13 Serial/Parallel Adapter: \n: \
      ErrorType A1: -n: LastI/O X1: \
      LineStatus D4: PrinterStatus B0.1:
- U11
- U12
```

To update an error template, you need to create a new template with the same error identifier but with a version and release number (*VV.RR* field) greater than the version and release number of the error template that you want to update. When **errupdate** merges the `.err` file into the master template file, it will replace the old template with the updated one.

Once you have created a file containing new or updated template definitions and/or template deletions, you can use **errupdate** to process it. In the following example, the template definitions and deletions are contained in a file called **newtemps.err**:

```
errupdate  newtemps
```

errdead

Extracts hardware error information from memory or from a dump file. This is used if the error daemon was not running or did not have a chance to log the error before a system failure. To use this command, invoke **errdead** with the name of a file containing dump information:

```
errdead mydump
```

Using the Error Log Subroutines

To use the error log subroutines, you need to understand the relationship between error IDs and error identifiers. You also need to know how to identify the type of an error.

Error IDs are 3 bytes long, and are created using hexadecimal digits. When you use the error log subroutines, you must specify the error ID of the program that generates the error entry. The pre-defined error IDs can range from 0x010000 to 0x050F0F. User-defined error IDs can range from 0x060000 to 0x060F0F. There is a special set of user-defined error IDs for hardware error entries. These error IDs can range from 0x010F00 to 0x010F0F. Note that the second and third bytes cannot use values greater than 0x0F for any error ID.

The error ID is used to categorize the error entries into appropriate groupings. The first digit is the ***class***; the second is the ***subclass***, and the third is the ***mask***. The digits are hierarchical in that a class consists of zero or more subclasses and a subclass consists of zero or more masks.

The class digits range from 0x01 to 0x06. Each value represents a general category of errors. The categories are assigned as follows:

- 0x01 — Hardware
- 0x02 — Software
- 0x03 — IPL/Shutdown
- 0x04 — General System Condition
- 0x05 — Not Available
- 0x06 — User-Defined.

The subclass and mask digits range from 0x00 to 0x0F. They are used to further subdivide the category represented by the class digit. Classes 0x01 through 0x05 may have pre-defined subclasses and masks. Class 0x06 has no pre-defined subclasses or masks because it is user-defined.

When you format a file containing error entries, you have the option of specifying which class of errors should be formatted. You can also specify the subclass within the class and the mask within the subclass. Note that when you format error entries, you do not use the error ID. Instead, you use an ***error identifier***. Error identifiers are similar to error IDs, except they consist of characters instead of hexadecimal digits and an alphabetic letter is used to specify the class instead of a digit. The letter is the first letter of the first word describing the class.

For example, if the error ID of an error entry is 0x010102, the error entry is in the **Hardware** class. The error identifier for this type of error entry would be H12. An error entry in the **Software** class might have an error ID of 0x020303. The error identifier would be S33. Thus, An error entry in the **User-defined** class would have an error identifier that began with the letter U.

Note: You should avoid defining error IDs that have a subclass or mask value of 0x00. This is because the error formatter uses the character 0 as a pattern-matching character. For example, to format all hardware error entries, regardless of subclass or mask, you

would specify H00 when you invoke the **errpt** command. The use of 0 within an error ID limits your selection options when you format the error entries.

When you design a program, you must decide what constitutes an error condition and what actions should take place if an error does occur. You decide in advance what the error ID will be for any error entries generated by a specific component. You must also specify the ***type*** of error entry that is being generated. There are several pre-defined types of error entries. Your component must contain an algorithm for deciding which type of error has occurred. Once the type is identified, it is used as input data to the error log subroutines.

The types of error entries are defined as follows:

- Permanent
 These are errors that are severe enough to prevent successful completion of an operation.
- Temporary
 These are errors that require an operation to be retried a defined number of times before being successfully completed.
- Information
 These are not necessarily errors. A component may generate an error entry of this type if an unusual condition occurs.
- Counters
 These error entries are generated by device driver components. Certain device drivers are able to generate retries if an operation is not successful on the first attempt. They use counters to monitor the number and cause of retries and contain algorithms that decide when these counters should be sent to the error log.
- Abbreviated Error Entries
 If the system fails before storing an error entry in the error log, it writes an abbreviated entry into non-volatile storage. When the system is started following such a failure, it writes this abbreviated entry into the error log.
- Expert Analysis Appended.
 These error entries contain information from the Error Log Analysis routine.

The data in an error entry is entirely dependent on the component generating the error entry. There is no defined limit on the size of an error entry. How you organize the information required to generate an error entry depends on which error log subroutine you use. Both **errunix** and **errsave** use an integer length field. This field specifies the length plus one (in words) of the error entry data. They are briefly defined below:

errunix Generates an error entry from an application and requires two input parameters. The first is the address of a buffer containing the 3-byte error ID, the 1-byte type number, an integer length field, and the error data for the error entry. The second is the length of the buffer.

errsave Generates an error entry from a kernel component and requires two input parameters. The first is the address of a buffer containing the 3-byte error ID, the 1-byte type number, an integer length field, and the error data for the error entry. The second is the length of the buffer.

_errvrm Generates an error entry from a VRM component and requires one input parameter. See the definition of the **_errvrm** subroutine for information on creating error entries within VRM components.

An example error point for application level components is shown in Figure 7-9. Notice how the input data structure used by the error log subroutine is organized and how the required data is placed in it. Also, the error ID for the trace entry is 0x060104. Thus, the error template used to format the error entry would use an error identifier of U14.

```
#include <sys/erec.h>

main(argc, argv)
   int argc;
   char **argv;
{
   extern int errunix();
   int error=0;
   struct error_struct   /* this is an input parameter for errunix */
   { union
      { struct
         {
            char e_class;
            char e_subclass;
            char e_mask;
            char e_type;
         } e_csmt;
         int csmt;
      } e_id;
      int e_len;       /* length + 1 of error entry data in words */
      char e_data[40];
   } error_data;

   #define CLASS     error_data.e_id.e_class
   #define SUBCLASS  error_data.e_id.e_subclass
   #define MASK      error_data.e_id.e_mask
   #define TYPE      error_data.e_id.e_type
```

Figure 7-9 (Part 1 of 2). Example Program Fragment Using errunix Subroutine.

```
   /* Check for errors and increment error flag if an error is found. */
   if (error_check() < 0)           /*  User-defined function */ /
     error++;

   /* if the error flag is positive, generate an error entry. */
   if (error)
   {
      CLASS    = E_USER;    /* E_USER is predefined as 0x06. */
      SUBCLASS = 0x01;
      MASK     = 0x04;
      TYPE     = E_TMP;     /* E_TMP is pre-defined as 0x40 */

      /* Collect the error data and put it in error_data.e_data */
      fill_in(error_data.e_data);   /*  User-defined function */ /

      /* Perform the subroutine call */
      if (errunix(&error_data, sizeof(struct error_struct) ) < 0)
      {
         fprintf(%s: Cannot log error. errunix failed.-n", argv[0]);
         exit(-1);
      }
   }
   .
   .
   .
}   /* end main */
```

Figure 7-9 (Part 2 of 2). Example Program Fragment Using errunix Subroutine.

Figure 7-10 creates an error entry from within a kernel component. Note that the user-defined subroutine **fill_in** must define the error ID and error data. The method is similar to that used in Figure 7-9 on page 7-32; therefore, it is not repeated for this example:

```
#include <sys/erec.h>

main(argc,argv)
    int argc;
    char **argv;
{
   struct errinfo *errptr;

   if (error)

   /* collect error data and call errsave */
   {
      fill_in(errptr);   /* User-defined subroutine */
      errsave(errptr, sizeof(struct errinfo));
   }
}   /* end main */
```

Figure 7-10. Example of a Program Fragment Showing Use of errsave Subroutine.

Creating Error Templates

The error formatter uses ***error templates*** to determine how the data contained in error entries should be formatted. All error templates are stored in the master template file, **/etc/errfmt**. After you create some new or updated error templates, place them in a file that can then be merged into the master template file using the **errupdate** command.

Error templates contain three required fields and zero or more data description fields. The required fields consist of the error identifier, the version and release number, and the error label. The data description fields contain formatting information for the error entry data and can be repeated as many times as is necessary to format all of the error data in the error entry.

Error entries are formatted and written to standard output one entry at a time. For each entry, the error formatter performs the following operations:

1. Locates the error template whose error identifier corresponds to the error ID in the error entry. If the error entry has an error ID of 0x010201, the corresponding error template would have an error identifier of H21.
2. Writes the time the entry was generated. This field is placed in the error entry header by the error subroutine.
3. Writes the PID, IODN, and IOCN contained in the error entry header. These fields are placed in the error entry header by the error subroutine.
4. Writes the error label contained in the format template. You can use the actual error identifier, or you can create a name up to 8 characters. The error formatter will write whatever is placed in the field.
5. Writes the error data according to the *data_descriptors* in the error template, if any are present. Any data that occurs after the last *data_descriptor* in the template is ignored. A *data_descriptor* does the following:
 a. Writes the data label (*d_label*), if it is specified.
 b. Writes the actual data according to the *format* field and the optional match fields. You can define match fields that will replace data values with descriptive labels or change to a different *data_descriptor* depending on the current data value.

Figure 7-11 on page 7-36 shows the syntax used to define an error template. Figure 7-12 on page 7-37 defines each of the fields in an error template.

Error template lines can be as long as you need. You can continue an error template on another line by adding the \ (backslash) character to the end of the line where the split occurs. Note that \ is not a substitute for a required blank. A blank is also required after a colon or comma.

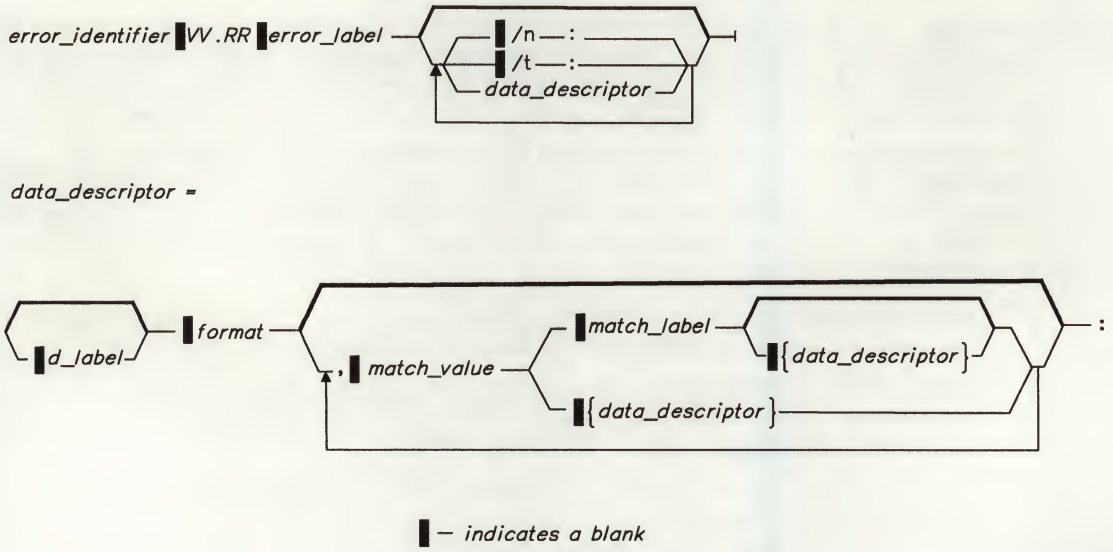


Figure 7-11. Error Template Syntax

Field	Description
error_identifier	The 3-character error identifier of this template.
VV.RR	A number representing the version and release level of this template.
error_label	A field of up to 14 characters that describes the error entries that use this template. This appears in the summary header list under the `Subclass` heading.
\n	A special string used to start output data a new line.
\t	A special string used to start output data after one tab unit.
d_label	A field of characters that describes the error entry data.
format	An alphanumeric code that defines the format of the error data.

Code	Format of Data
A*m*	ASCII string of *m* characters.
B*m*	Binary string of *m* bytes.
B*m.n*	Binary string of *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
D2	Decimal short integer.
D4	Decimal long integer.
F4	Floating-point number of type *float*, rounded to four places.
F8	Floating-point number of type *double*, rounded to four places.
O*m*	Omit (do not write) the next *m* bytes.
O*m.n*	Omit (do not write) the next *m* bytes and *n* bits. If this format leaves some unformatted bits in the current byte, they are not written unless this format is followed by another format that specifies bits.
U2	Unsigned decimal short integer.
U4	Unsigned decimal long integer.
X*m*	Hexadecimal number of *m* bytes.

match_value	A value with a data type the same as the *format* field. If you specify a *match_value*, you must also specify a *match_label* or *data_descriptor*.
match_label	A character field up to 120 characters that replaces output data that matches a *match_value* field.
data_descriptor	A field containing formatting information for a portion of the output data.

Figure 7-12. Fields in an Error Template

Replacing Values in the Output Data

The *match_value* field provides a function that is similar to a switch statement in programming. You can use this function to do either or both of the following:

- Replace an output data value with a character string that describes the data. For example, you can replace a numeric error code with a description of that error. The replacement character string is called a *match_label.*
- Change the data descriptor used to format the output data. For example, you can design error points that will record different data at different times, depending on the state of your machine or program. You can then define a set of *match_values* with a different *data_descriptor* for each type of data.

You can specify any number of *match_value* fields for a particular output data value as long as each field is accompanied by a *match_label* and/or a *data_descriptor* field. If the output data value matches *match_value*, the error formatter uses the *match_label* and/or *data_descriptor* associated with the *match_value.*

You can use the special string * as a *match_value.* This will match any output data value in the current data field. When the error formatter finds this string, it does not check any *match_values* that occur after this string. Therefore, you should use this string as the last *match_value* in a list to provide a default action if none of the other *match_values* are matched.

If a new *data_descriptor* follows the *match_value*, the error formatter uses the formats from the new *data_descriptor* until one of the following occurs:

- It has formatted all of the error entry data.
- It has used all of the defined *data_descriptors* for the current output data.

The error formatter then looks for a new *data_descriptor* field to define the format for the rest of the data in the error entry. If a new *data_descriptor* is not found, it writes the data literally according to the format given in the *format* field of the last *data_descriptor.*

Note: The error formatter compares a *match_value* field and the current output data as two character strings. For real numbers (*F* format), it uses the **printf** subroutine to round the data to four decimal places. If you plan to use a *match_value* field with output data formatted as a real number, the *match_value* field should use the same precision, four decimal places.

Appearance of the Formatted Output Data

The following examples show you how to format data using a single *data_descriptor*. The data is defined as 8 bytes consisting of IBM4341A. The *data_descriptor* fields have the following values:

d_label	PtrNode
format	A8
match_value	IBM4341A
match_label	Model4X

Each of the following examples shows a *data_descriptor* using a combination of *data_descriptor* fields and how the output data would look if it used that *data_descriptor*:

- If you just specify a *format* field, the *data_descriptor* would be:

  ```
  A8:
  ```

 The formatted error report would show the data as:

  ```
  IBM4341A
  ```

- If you specify *d_label* and *format* fields, but do not specify a *match_value* field, the *data_descriptor* would be:

  ```
  PtrNode A8:
  ```

 The formatted error report would show the data as:

  ```
  PtrNode=IBM4341A
  ```

- If you specify *format*, *match_value*, and *match_label* fields, but do not specify a *d_label* field, the *data_descriptor* would be:

  ```
  A8, IBM4341A Model4X:
  ```

 The formatted error report would show the data as:

  ```
  Model4X
  ```

- If you specify *d_label*, *format*, *match_value*, and *match_label* fields, the *data_descriptor* would be:

  ```
  PtrNode A8, IBM4341A Model4X:
  ```

 The formatted error report would show the data as:

  ```
  Ptrnode=Model4X
  ```

Error Template Example

This section shows you how an error entry using a sample error template would appear after being formatted. In the example, a hardware error entry was generated for the Serial/Parallel Adapter. The header data for the error entry includes the error ID, which in this example is 0x060103. Thus, the error identifier in the template is U13. The error entry data is listed below:

`ErrorType` The type of error encountered (1 ASCII character). The *format* for this data would be A1.

`LastI/O` The last character transmitted (1 hexadecimal digit). The *format* for this data would be X1.

`LineStatus` The status of the printer (1 decimal word). The *format* for this data would be D4.

`PrinterStatus`
A 1-bit flag that indicates if the printer is active (1) or inactive (0). The *format* for this data would be B0.1.

Figure 7-13 shows a sample error template for error entries with an error ID of 0x060103. Notice the newline descriptor on the first line of the template. This will cause the data to start on a new line.

```
U13 Ser/Par: \n: \
    ErrorType A1: \n: Last_I/O X1: LineStatus D4: PrinterStatus \
    B0.1:
```

Figure 7-13. Example of an Error Template for Error Entries with Error ID 613

Figure 7-14 on page 7-41 shows how an error entry using the template U13 would appear in a formatted error report. The first entry is from a different error. The second entry is the one that uses template U13.

```
                                ERROR LOG REPORT

Error log: /usr/adm/ras/errfile
-------------------------------------------------------------------------------
Date/Time            Class          Subclass       Type      Device  Cause
Thu May  9 16:54:42  Hardware       Diskette       Counters  DSKT    Hardware
IODN=0004       IOCN=0240
Base_Port_Address=000003F2
Dev_Name=DSKT
Internal_Dev_Type=D1015200
        Switchable_to_Coprocessor
        8_bit_device
        No._Interrupts=1
        Slot_Number=1
        Adapter_Type=52
        Port_Number=00
Bad_IO_Address_Mark

Bad_Count=4
Good_Count=74
Bad_Threshold=3
Ratio=0

-------------------------------------------------------------------------------
Date/Time            Class          Subclass       Type      Device  Cause
Thu May  9 16:54:45  Hardware       Ser/Par        Perm      Ser/Par Hardware
        IODN=7  IOCN=600
        Base Port Addr=0278
        Dev Name=Ser/Par
        Internal Dev Type=91002300
ErrorType=3
Last_I/O=44 LineStatus=0 PrinterStatus=0

===============================================================================
```

Figure 7-14. Example Output from the Error Formatter

Using the Dump Facilities

The dump facilities allow you to dump VRM data onto a dump diskette in the event of a system failure at the VRM level. Some of the basic terms are defined below:

dump data The data collected by the dump program. It is obtained from memory locations used by VRM components.

dump diskette
The diskette used to store the dump data. This must be a formatted, high-capacity diskette and it must be placed in diskette drive A before the dump is started. A dump operation is limited to the storage capacity of a single dump diskette.

component dump table
A structure used by VRM components to identify data structures that should be collected by the dump program. Each record in a component dump table identifies the name, length and location of a specific data structure.

dump table entry
A record in the master dump table that identifies the location of a component dump table. All VRM components that need to have special data collected by the dump program need to generate a dump table entry. Each entry contains the component ID, length, and location of a component dump table.

master dump table
A structure containing dump table entries generated by VRM components. The dump program uses this table to locate data structures that should be included in a dump.

How you use the dump facilities depends on what you want to accomplish:

- To start a dump operation, you should learn how to use the dump key sequence.
- To format the contents of a dump diskette or dump file, you should learn how to use the **dumpfmt** command.
- To create dump table entries in the master dump table, you should learn how to use the **_dmptbl** subroutine. All of the guide information for this subroutine is contained in *Virtual Resource Manager Technical Reference.*

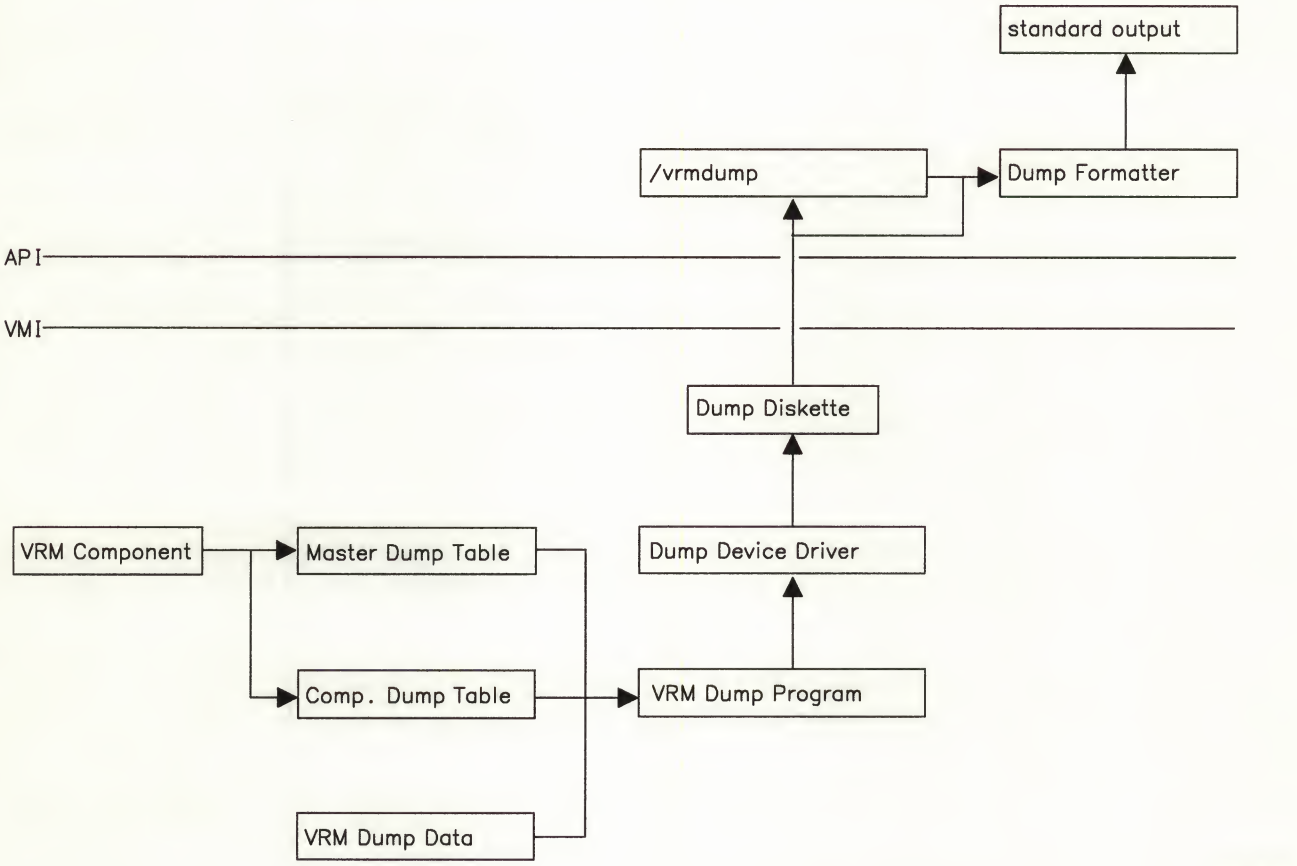


Figure 7-15. Dump Components

Figure 7-15 shows how data is passed between the various files and components that constitute the dump facilities. The lines labeled API, which stands for Application Program Interface, and VMI, which stands for Virtual Machine Interface, show the logical distinction between application programs, operating system kernel components, and the operating system VRM component. The lines connecting the files and components show where data comes from and where it goes.

The components and files in Figure 7-15 are described on the next page. They are explained in more detail elsewhere in this chapter in the appropriate sections.

The descriptions below start with the input to the dump program and end where the dump diskette or file is formatted and sent to standard output.

- VRM components generate master dump table entries using the **_dmptbl** subroutine. Only VRM programs are supported by this subroutine.
- The master dump table contains dump table entries that identify the component ID, length, and location of component dump tables. Your only interaction with the master dump table is through the **_dmptbl** subroutine.
- Component dump tables are created and maintained by individual VRM components. Each component dump table contains a list of entries that specify the name, length, and location of data structures that should be included in the dump. There is a 4-byte length field at the beginning of each component dump table. This field contains the overall length of the component dump table. Each entry in the table is 20 bytes long. Thus, a component dump table with two entries would be 44 bytes long.
- The VRM dump data is pre-defined and consists of various types of VRM data. This includes tables, control blocks, queue areas, error and trace buffers, virtual machine information, registers, and other data structures. You may not be able to understand all of this data. It is used by IBM to determine the cause of a system failure.
- The VRM dump program collects the VRM dump data plus any data specified in component dump tables and writes it onto a dump diskette. This program is always resident in memory and is invoked through the dump key sequence or by a routine below the VMI executing a signal error.

 When VRM dump is initiated, it sets the two-digit display on the system unit to `c6` to indicate that it wants to perform a dump. You can cancel the dump, or insert a dump diskette and press the dump key sequence that begins the dump operation.
- The dump device driver is used only by the dump program to write to the dump diskette. This device driver allows the dump program to operate without the use of any VRM device drivers. The VRM dump program is independent of any VRM resources.

 Because the dump data is written in dump/restore format, you can use the **restore** command to write the diskette data onto a file in the current directory. The file name of a restored dump data file is always *vrmdump*, but you can rename it if you wish.
- The dump formatter formats the data from the dump diskette or from a specified file into a readable format. If a file is not specified, it looks for the information in **/dev/fd0**, which is where the dump diskette should be. The formatted dump data is sent to standard output and can therefore be sent to the display screen, a file, or a printer.

Using the Dump Commands

There are only two dump commands. The first command is the key sequence that is used to start the dump program. You use this sequence when your system fails and c6 is displayed on the two-digit display on the system unit. The dump procedure is fully discussed in the *Problem Determination Guide*. The following steps outline the dump procedure:

1. Insert a formatted diskette into diskette drive A:. The diskette can have AIX Operating System or DOS format.
2. Press **Ctrl-Alt-Num8**. c9 is displayed on the two-digit display to indicate that the dump program has started.

The dump program runs until it completes the dump or encounters an error condition. The two-digit display shows the result of the dump upon termination. The possible two-digit display values are:

0 — Dump Successfully Completed
2 — Dump Diskette Full
3 — Dump Suspended Due to Recursive Call
4 — Dump Abended.

Before you can use the second command, **dumpfmt**, you have to get the system back in working order. Once you get the system restarted, you can use **dumpfmt** to format the dump data and send it to standard output.

The **dumpfmt** command invokes the dump formatter, which can be used in batch mode or interactive mode. In batch mode, all of the dump data appears in the report. In interactive mode, you select the data you want to format from a menu of options. If you do not specify batch mode using the **-a** flag, the formatter automatically comes up in interactive mode.

Example of the Dump Formatter Output

Figure 7-16 on page 7-46 shows how part of a dump diskette or dump file might appear after being formatted. It displays data in hexadecimal format with ASCII interpretation at the right. Non-printing hexadecimal values are shown as a period. Notice how each set of dump data structures has a different title. A name title (GP Register) indicates that the data is part of the standard VRM dump data. A number title (126) indicates that the data is obtained from a component dump table.

Note: The dump report shown in Figure 7-16 does not show all of the pre-defined VRM dump data. The dump program collects all of the pre-defined VRM dump data before it begins collecting the component dump table data. If you use the dump formatter in interactive mode, you can select which types of dump data you wish to format.

```
Tue Jan 03 12:00:00 1985

Failing Component: ""

Module Start Address: "00000000"

Module Offset Address: "00000000"

                    GP Register  Starting Address : "C0028C40"

 reg#_13
C0028C40   DD2E0040 69F0CC6B 04001920 00000BA0     | ...@ i..k .... ....|
C0028C50   0002CED8 F0000000 00028C20 0000B3E0     | .... .... .... ....|
C0028C60   701B0203 65100008 659ABE50 630D3013     | p... e... e..P c...|
C0028C70   71BBE2BA 31B3D402 000292A8 00008B54     | q... .... .... ...T|
C0028C80   BE526DD2 300DDD9E 0040001C 6400E549     | .Rm. .... .@.. d..I|
C0028C90   64349141 6840E289 6910E298 B49A091B     | d..A h@.. i... ....|
C0028CA0   E29A0204 E21A311B 000B63A9 E213311B     | .... .... ..c. ....|
C0028CB0   62A4BE52 703B6DD2 303D319D 302BDD4E     | b..R p;m. .=.. ..N|
C0028CC0   004031F7 8DFFB87A 62656500 302BDD4E     | .@.. ...z bee. ....|
     .
     .
     .
                    126  Starting Address : "0"

 Header
00000000   54686973 64617461 63616D65 66726F6D     | This data came from|
00000010   636F6D70 6F6E656E 74206475 6D702074     | comp onen t.du mp.t|
00000020   61626C65 77697468 20494420 20313236     | able with .ID. .126|
00000030   701B0203 65100008 659ABE50 630D3013     | p... e... e..P c...|
00000040   BE526DD2 300DDD9E 0040001C 6400E549     | .Rm. .... .@.. d..I|
00000050   71BBE2BA 31B3D402 000292A8 00008B54     | q... .... .... ...T|
```

Figure 7-16. Example of Output from the Dump Formatter

Chapter 8. Debugging Programs

CONTENTS

About This Chapter 8-4
The dbx Symbolic Debugger 8-5
dbx Command Summary 8-6
Starting dbx 8-6
Setting and Deleting Breakpoints 8-6
Running Your Program from dbx 8-7
Tracing Program Execution 8-7
Ending Program Execution 8-8
Displaying the Source File 8-8
Printing and Modifying Variables, Expressions, and Types 8-9
Multiprocess Debugging 8-9
Procedure Calling 8-10
Signal Handling 8-10
Machine Level Debugging 8-10
Debugging Environment 8-11
Invoking the dbx Debugger 8-11
Other Invocation Options 8-13
Controlling Program Execution 8-14
Setting and Deleting Breakpoints 8-14
Running Your Program 8-16
Separating dbx Output from Program Output 8-17
Tracing Execution 8-18
Signal Handling 8-19
Calling Procedures 8-20
Displaying a Stack Trace 8-20
Printing Variables and Expressions 8-21
Displaying and Modifying Variables 8-21
Scoping of Names 8-22
Operators Allowed in Expressions 8-23
Type Checking in Expressions 8-23
Folding Variables to Lower- and Upper-Case 8-24
Special Debugger Variables to Change Print Output 8-25
Displaying and Manipulating the Source File 8-26
Changing the Source Directory Path 8-26
Displaying the Current File 8-26
Changing the Current File or Procedure 8-27
Debugging Programs Involving Multiple Processes 8-28

Debugging at the Machine Level 8-30
Machine Registers 8-30
Examining Memory Addresses 8-31
Running Your Program at the Machine Level 8-32
The dbx Debugging Environment 8-33
The Alias Facility 8-33
Changing the dbx Prompt 8-34
Customizing Your Environment with .dbxinit 8-34
Reading dbx Commands from a File 8-35
Running Shell Commands from dbx 8-35
Getting Help 8-36
sdb to dbx Equivalencies 8-36
Using dbx with X-Windows 8-37
The xdbx Window Environment 8-37
xdbx Modes 8-40
Using the Mouse 8-41
Changing the xdbx Environment 8-42
Attaching a Process 8-45
Debugging Multiprocess Programs 8-45
The sdb Program 8-47
sdb Command Summary 8-48
Start/Stop sdb 8-48
Setting Breakpoints 8-48
Setting Trace Options 8-48
Running the Program from sdb 8-49
Single Stepping 8-49
Displaying Information 8-50
Changing Values of Program Variables 8-50
Changing Position in the File 8-50
Reference Metacharacters 8-51
Using the Program 8-51
Displaying a Stack Trace 8-52
Examining Variables 8-53
Referencing Arrays, Pointers, and Structures 8-55
Displaying and Manipulating the Source File 8-55
Displaying the Current File 8-56
Changing the Current File or Procedure 8-56
Changing the Current Line 8-56
Controlling Program Execution 8-57
Setting and Deleting Breakpoints 8-57
Running the Program 8-58
Calling Procedures 8-60
Debugging Assembler Language 8-60
Displaying Assembler Language Statements 8-60
Manipulating Registers 8-61
An Example of a Debug Session 8-61

The adb Program 8-64
adb Command Summary 8-64
Starting with a Program File 8-66
Starting with a Core Image File 8-67
Starting with a Data File 8-67
Starting with the Write Option 8-68
Using a Prompt 8-68
Leaving adb 8-69
Displaying Instructions and Data 8-69
Forming Addresses 8-70
Choosing Data Formats 8-71
Displaying an Address (=) 8-76
Using Integers in Expressions 8-77
Using Symbols in Expressions 8-77
Finding the Current Address 8-79
Using Operators in Expressions 8-83
An Example of Simple Formatting 8-85
Debugging Program Execution with adb 8-88
Executing a Program 8-89
Setting Breakpoints 8-90
Displaying Breakpoints 8-91
Deleting Breakpoints 8-92
Continuing Execution 8-92
Single-Stepping a Program 8-92
Stopping a Program with Interrupt and Quit 8-93
Killing a Program 8-93
Displaying the C Stack Backtrace 8-94
Displaying External Variables 8-95
An Example: Tracing Multiple Functions 8-96
Using the adb Memory Maps 8-100
Displaying the Memory Maps 8-100
Changing the Memory Map 8-101
Combining Commands on a Single Line 8-102
Creating adb Scripts 8-103
Setting Output Width 8-107
Setting the Maximum Offset 8-107
Setting Default Input Format 8-108
Using AIX Operating System Commands 8-108
Computing Numbers and Displaying Text 8-109
An Example: Directory and Inode Dumps 8-110
Patching Binary Files 8-113
Locating Values in a File 8-113
Writing to a File 8-113
Making Changes to Memory 8-114

About This Chapter

This chapter describes the program debug programs supplied with the operating system: **dbx**, **sdb**, and **adb**. The first part of the chapter explains how to use **dbx** to help find and fix program logic problems (***bugs***) and how to use **dbx** in an **xdbx** windowing environment.

The second part of this chapter explains how to use **sdb** to help find and fix program logic problems. It shows how to use the **sdb** commands and discusses the idea of a controlled program testing environment. Also included is an **sdb** session example.

The third part of the chapter explains how to use the **adb** program to help find and fix logic problems in C and assembler language programs. In particular, it explains how to:

- Start and stop **adb**
- Display program instructions and data
- Run, set breakpoints, and single-step a program
- Patch program files and memory.

The third part also illustrates techniques for debugging C programs and explains how to display information in non-ASCII data files.

The dbx Symbolic Debugger

Using the dbx symbolic debugger you can:

- Debug programs at either the source language or assembler language level
- Set breakpoints at selected statements and machine instructions with conditions for activation
- Run a program one line or one instruction at a time
- Access variables symbolically and display them in the correct format
- Display or modify the contents of machine registers, variables, and memory
- Examine the source text using simple search functions or your own editor
- Debug processes and subprocesses that contain **fork** and **exec** system calls
- Interrupt and examine a program that is already in progress
- Trace execution of a program by line, instruction, routine, or variable
- Display expressions using a wide range of operators
- Print a list (stack trace) of the active routines and their parameters
- Print declarations of variables, along with their fully qualified names
- Customize your interface to the debugger with command aliases
- Customize your debugging environment with an initialization file
- Call program or diagnostic routines directly from the debugger
- Invoke your choice of editors and shell commands during the debugging session
- Open a virtual terminal to separate debugger interaction from your program
- Issue commands from either standard input or a named file
- Modify the directory list from which to search for source files.

dbx Command Summary

The following is a list of **dbx** commands with a short description of each. For more information, see *AIX Operating System Commands Reference.*

Starting dbx

dbx [*filename* [*corefile*]**]**
Starts **dbx** using *filename* as the program.

dbx -r *filename*
Runs *filename* immediately, and enters **dbx** only if the program terminates unsuccessfully.

dbx -a *pid*
Attaches **dbx** to a process already in progress with process ID = *pid.*

dbx -I *dir* **[***filename* **]**
Adds *dir* to the list of directories that are searched when looking for a source file.

dbx -c *cmdfile* **[***filename***]**
Runs the **dbx** commands in *cmdfile* before reading from standard input.

Setting and Deleting Breakpoints

stop at *linenumber* [**if** *condition*]
Stops execution at *linenumber.*

stop in *procedure/function* [**if***condition*]
Stops execution when the procedure is reached.

stop *variable* [**in** *procedure/function*] [**if***condition*:]
Stops execution when the value of *variable* is modified.

stop if *condition*
Stops execution when the condition is true.

status [>*filename*]
Displays the currently active breakpoints and trace events.

delete *event-number* [, *event-number,* . . .]
Removes the breakpoint and trace events associated with the event numbers.

clear *linenumber*
Removes all breakpoints at the given source line.

cleari *address*
Removes all breakpoints at the given memory location.

delete all
Removes all breakpoints and trace events.

Running Your Program from dbx

run [*args*] [<*filename*] [>*filename*] [> >*filename*] [2>*filename*] [2> >*filename*] [>&*filename*] [> >&*filename*]
Starts executing program, passing *args* as command line arguments. Redirection of input, output, and errors occurs in the usual manner.

rerun [*args*] [<*filename*] [>*filename*] [> >*filename*] [2>*filename*] [2> >*filename*] [>&*filename*] [> >&*filename*]
Same as **run**, except the default argument list is the previous argument list specified in a **run** or **rerun**.

cont [*integer* | *signalname*]
Continues execution of the program from where it stopped. If a signal is specified, the process continues as though it received the signal.

step [*num*]
Executes one or *num* program source line(s).

next [*num*]
Executes up to the next or the next *num* of program source line(s). The **next** subcommand does not stop execution during execution of called procedures.

return [*procedure*]
Continues execution until a return to *procedure* is executed, or until the current procedure returns if *procedure* is not specified.

stepi [*num*]
Executes one or *num* machine instruction(s).

nexti [*num*]
Executes up to the next or next *num* machine instruction(s). The **nexti** command does not stop execution during execution of called procedures.

skip [*num*]
Continues execution until the end of the program or until *num* + 1 breakpoints.

goto *source_line_number*
Changes the next line executed to *source_line_number*.

gotoi *address*
Change the program counter to *address*.

Tracing Program Execution

trace
Prints each source line executed in the program.

trace in *procedure/function* [**if** *condition*]
Prints each source line executed while *procedure/function* is active.

trace *source_line_number* [**if** *condition*]
Prints the source line when execution reaches *source_line_number*.

trace *expression* **at** *linenumber* [**if** *condition*]
Prints value of *expression* at *linenumber*.

trace *variable* **[in** *procedure/function***]** **[if** *condition***]**
Prints tracing information when the value of *variable* changes.

tracei [*variable***]** **[at** *address***]** **[if** *condition***]**
Traces one program instruction.

tracei *address* **[if** *condition***]**
Prints tracing information when the contents of *address* changes.

watch *variable* **at** *linenumber* **[if** *condition***]**

watch *variable* **[in** *procedure/function***]** **[if** *condition***]**
The same as **trace** except running under **xdbx**. Under **xdbx**, information printed is placed in a special **watch** window.

where [**>***filename*]
Displays a stack trace of procedures and parameters.

Ending Program Execution

quit
Exits from the debugger and the program.

Displaying the Source File

use *directory1 directory2* . . .
Sets the list of directories to search for source files to the directories listed in this command.

list *line-number expression* **[,** *linenumber***]**
Displays the lines in the current source file from the first line number to the second line number, inclusive.

list *procedure/function*
Displays lines surrounding the first line of *procedure*.

listi *procedure/function*
List instructions from the specified *procedure* or *function*.

listi at *source_line_number*
Lists instructions beginning at the specified *source_line_number*.

listi [*address***[,***address***]]**
List instructions from the first address to the second address inclusive.

file [*filename*]
Changes the current file to *filename*. If no file is specified, then the current source filename is displayed.

func [*procedure/function*]
Changes the current function. If no procedure or function is specified, then the current function is displayed.

/ *regular expression* **[/]**
Searches forward in the current source file for the given expression.

? *regular expression* [?]
Searches backward in the current source file for the given expression.

edit [*filename*]
Invokes an editor for *filename* or the current source file if none is specified.

edit *procedure/function*
Invokes an editor on the file containing the specified procedure or function.

move *source_line_number*
Changes the next line displayed to *source_line_number*.

Printing and Modifying Variables, Expressions, and Types

print *expression* [, *expression* . . .]
Prints the values of the given expressions.

dump [*procedure*] [> *filename*]
Prints the names and values of the variables in *procedure*, or the current procedure if none is specified.

assign *variable* = *expression*
Assigns the value of *expression* to *variable*.

case [default | mixed | lower | upper]
Changes the way that **dbx** interprets symbols.

whatis *name*
Displays the declaration of the given name, which may be qualified with block names as above.

which *identifier*
Displays the full qualification of the given identifier (the outer blocks that contain the identifier).

whereis *identifier*
Displays the full qualification of all symbols that match the given identifier. Order of output is not meaningful.

Multiprocess Debugging

multproc [on | off]
Enables or disables multiprocess debugging. The initial value is **off**. Issuing the command with no arguments displays the current status.

Procedure Calling

call *procedure* (*parm1, parm2, . . .*)
Executes the object code associated with *procedure.*
print *procedure* (*parm1, parm2, . . .*)
Executes the object code associated with *procedure* and prints the value returned by *procedure.*

Signal Handling

catch [*integer* | *signalname*]
Starts trapping a signal before it is sent to your program. Signals can be specified by number or by name. Signal names are not case sensitive and the SIG prefix is optional. Use this command with no arguments to list trapped signals.
ignore [*integer* | *signalname*]
Stops trapping a signal before it is sent to your program.

Machine Level Debugging

address, address/ [*mode*] [>*filename*]
Prints the contents of memory starting at the first address and continuing to the second address. The mode specifies how memory is to be printed.
address / [*count*] [*mode*] [>*filename*]
Prints the contents of memory starting at the first address and continuing up to the second address, or until *count* items have been displayed. The mode specifies how memory is to be printed.
stopi [*variable*] [**at** *address*] [**if** *condition*]
Halts execution when the machine instruction is reached.
stopi *address* [if *condition*]
Stops execution when the machine instruction is reached.
tracei [*variable*] [**at** *address*] [**if** *condition*]
Traces execution of the specified machine instruction.
tracei *address* [**if** *condition*]
Traces execution of the specified machine instruction.
stepi [*num*]
Executes one or *num* machine instructions.
nexti [*num*]
Executes up to the next or a specified *num* of machine instruction. The **nexti** command does not follow branch and link instructions; it continues until execution returns to the next instruction.

registers [>*filename*]
Displays the values of the machine registers and the current instruction.
gotoi *address*
Change the program counter to *address*.

Debugging Environment

alias [*aliasname* [*commandname*]]
Builds a user alias from the arguments specified.
alias *aliasname* **"commandstring"**
Builds a user alias.
alias *aliasname* **(***parm1, parm2, . . .* **)** *"commandstring"*
Builds a user alias from the arguments specified. Specifying **alias** with no arguments displays the current aliases in effect. Specifying **alias** with one argument displays the replacement string associated with that alias.
unalias *aliasname*
Removes an alias with the given name.
prompt *"promptstring"*
Changes the **dbx** prompt to *promptstring*.
help
Prints a brief description of common **dbx** commands.
source *filename*
Reads **dbx** commands from *filename*.
sh *commandline*
Passes *commandline* to the shell for execution. If *commandline* is missing, the shell is entered for use until it is exited, at which time control returns to **dbx**. The **SHELL** environment variable determines which shell is used.

Invoking the dbx Debugger

There are three ways to start a debug session with **dbx**:

dbx [*options*] [*objfile* [*coredump*]]
This is the usual way to invoke **dbx**. If *objfile* is not specified, then **dbx** asks for the name of the object file to be examined. The default is **a.out**. If either *coredump* is specified or a file named **core** exists in the current directory, then **dbx** reports the location where the program faulted. Examining variables, registers, and memory is applicable to the core image until execution of *objfile* begins. **dbx** prompts for commands at that time.
dbx -r [*options*] [*objfile*]
Use the **-r** option if you only want to enter the debugger if your program terminates abnormally. If *objfile* terminates successfully, **dbx** exits. Otherwise,

the reason for termination is reported and you are offered the option of entering the debugger or letting the program fault.

dbx [*options*] **-a** *pid*
Use the **-a** option if you want to debug a process that is already in progress. To use this option, you must have permission to kill the process that has a process ID of *pid*. **dbx** interrupts the process if access is allowed, determines the full path name of the object file, reads in the symbolic information, and then prompts for commands. Interaction at that point is no different from if you invoked **dbx** to begin execution of the process.

The following program examples show the generation and use of an executable program called `samp`. When the program runs, it produces a ***bus error*** and stops. The commands show how to start and end the **dbx** program.

In the first example, the program `samp` has run, faulted, and caused a core image to be written. **dbx** can be used to examine the core-image dump and attempt to determine the location of the error within the program as follows:

```
$ cc -g samp.c -O samp
$ samp
Bus error - core dumped
$ dbx samp
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information  . . .
[using memory image in core]
    25      x[i] = 0;
(dbx) quit
```

In the second example, **dbx** can be used to examine the state of the process in memory even though a core-image dump is not taken:

```
$ dbx -r samp
Entering debugger  . . .
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information  . . .
bus error in main at line 25
    25      x[i] = 0;
(dbx) quit
$
```

The third way of invoking **dbx** is used to attach to a process that is already running. Consider the following program, `looper.c`, which has a misplaced semicolon:

```
for (i = 0; i > 10; );
   printf(x[i]);
```

Compile the program with the **-g** option to get symbolic debugging capability:

```
$ cc -g looper.c -o looper
```

Run the following program from the command line:

```
$ looper
```

Seeing that your program is not terminating as expected, you want to debug it. To attach to `looper`, you must determine the ID number associated with the process. You must have another virtual terminal open if you did not run `looper` as a background process. From the other virtual terminal, run the system command **ps -u** *userid*, where *userid* is your login ID. All active processes that belong to you are displayed:

```
PID      TTY  TIME   COMMAND
68   console  0:04   sh
467    hft/3 10:48   looper
```

The process ID associated with `looper` is 467. Attach **dbx** to `looper`:

```
$ dbx -a 467
Waiting to attach to process 467  . . .
Determining program name  . . .
Successfully attached to /u/user/looper  . . .
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information  . . .
(dbx)
```

You can now query and debug the process as if it had been originally started with **dbx**.

Other Invocation Options

dbx -I *dir1* **-I** *dir2* . . . [*options*] **[***objfile* [*coredump*]]

Adds *dir1*and *dir2* to the list of directories that are searched when looking for a source file. Normally **dbx** looks for source files in the current directory and in the directory where *objfile* is located. If your source is in `/u/user/src` and `/u/group/src`, and *objfile* is in `/u/user/bin`, then the **-I** option should be used so that **dbx** can find the source automatically:

```
$dbx -I /u/user/src -I /u/group/src objfile
```

The **use** command can be used for this purpose once **dbx** is invoked. The **use** command differs because it resets the list of directories to be searched, whereas the **-I** option adds a directory to the list.

dbx -c *cmdfile* [*options*] [*objfile* [*coredump*]]
Executes the **dbx** commands in *cmdfile* before accepting commands from standard input. The **source** command can be used for this purpose once **dbx** is invoked.

Controlling Program Execution

dbx allows you to set breakpoints (stopping places) in the program. After entering **dbx**, you can specify which lines or addresses are to be breakpoints, and then run the program with **dbx**. When the program reaches a breakpoint, it halts and reports that it has reached a breakpoint. You can then use **dbx** commands to examine the state of your program.

An alternative to setting breakpoints is to run your program one line or instruction at a time, a procedure known as ***single stepping***. In the following sections we discuss how to set and delete breakpoints, begin program execution, and control program execution.

Setting and Deleting Breakpoints

Use the **stop** command to set breakpoints in **dbx**. There are four variations of the **stop** command for programs compiled with the debug flag (usually **-g**):

stop at *linenumber* [**if** *condition*]
A *source_line_number* is an optional file name and a : (colon) followed by a line number. For example, `(dbx) stop at "hello.c":23` or `(dbx) stop at 23`. The optional **if** *condition* specifies that execution should be halted at the specified line number if the condition is true when the line number is reached. Line numbers are relative to the beginning of the source file. A condition is an expression that evaluates to true or false. Valid expressions are described further in "Printing Variables and Expressions" on page 8-21.

stop in *procedure/function* [**if** *condition*]
Use this version of **stop** to stop the program at the first executable line number in a procedure of function. For example, `(dbx) stop in main`.

stop *variable* [**if** *condition*]
Use this version of **stop** to stop the program when the value of *variable* changes. For example, `(dbx) stop x`.

stop if *condition*
Use this version to stop the program whenever *condition* evaluates to true. For example, `(dbx) stop if (x > y) and (x < 20000)`.

After any of the preceding commands, **dbx** responds with a message reporting the event it has built as a result of your command. The message includes the event ID associated with your breakpoint along with an interpretation of your command. The syntax of the interpretation might not be exactly the same as your command. The following are examples:

```
(dbx) stop in main
[1] stop in main
(dbx) stop at 19 if x == 3
[2] if x = 3 { stop } at "hello.c":19
```

The numbers in the brackets are the event identifiers associated with the breakpoints. When the program is halted as the result of one of the events, the event identifier is displayed along with the current line to show what event caused the program to stop. The events you create exist with internal events created by **dbx**, so event identifiers might not always be sequential.

Use the **status** command to display all current events. You can redirect output from **status** to a file. Each event is displayed in the same form as it was generated. The **delete** command eliminates events by event identifier.

The **delete** and **clear**. commands remove breakpoints. The **clear** command deletes breakpoints by line number. Use **delete all** to remove all breakpoints and trace events. The following examples show how to display the active events and remove them:

```
(dbx) status
[1] stop in main
[2] if x = 3 { stop } at "hello.c":19
(dbx) delete 1

(dbx) status
[2] if x = 3 { stop } at "hello.c":19
(dbx) clear 19
(dbx) status
(dbx)
```

To set breakpoints in programs or routines that have not been compiled with the debug flag, see "Debugging at the Machine Level" on page 8-30.

Running Your Program

The **run** command starts your program. The command format is:

run [*args*] [< *inputfile*] [> *outputfile*] [> > *outputfile*] [*2*> *errfile*] [*2*> > *errfile*] [> & *outerrfile*] [> > & *outerrfile*]

This tells **dbx** to begin running *objfile*, passing *args* just as if they were typed on the shell command line. Input can be redirected from *inputfile* and to *outputfile* by using the preceding angle-bracket notation. The **rerun** command has the same form as **run**; the difference is that if no arguments are passed, the argument list from the previous execution is used. After your program begins, it continues until one of the following:

- The program reaches a breakpoint.
- A signal occurs that is not ignored. For example, **INTERRUPT** or **QUIT**.
- A multiprocess event occurs while multiprocess debugging is enabled.
- The program completes.

In each case, **dbx** receives control and displays a message explaining why the program stopped.

There are several ways to continue the program once it is stopped:

cont [*integer* | *signalname*]
: Continues the program from where it stopped. If a signal is specified, the process continues as though it received the signal. Signals can be specified by number, name, or name without the **SIG** prefix. Signal names can be either lower- or upper-case. The following commands are equivalent:

```
cont 3
cont SIGQUIT
cont quit
```

detach [*integer* | *signalname*]
: Continues the program from where it stopped, exiting the debugger. If a signal is specified, the process continues as though it received the signal. After your program completes, control is passed to the shell. This is useful after you have patched your program and want to continue without the debugger.

step [*num*]
: Runs one or more source lines.

next [*num*]
: Runs up to the next or a specified *num* of source line(s).

A common method of debugging is to step through your program one line at a time. The **step** and **next** commands are for that purpose. The distinction between these two commands is apparent only when the next source line to be run involves a call to a

function or procedure. In this case, the **step** command stops in the subroutine; in contrast the **next** command runs until the subroutine has finished and then stops at the next instruction after the call.

With **step**, the debugger stops after each machine instruction to see if the program has reached any line numbers. With **next**, the debugger sets an internal breakpoint at the address associated with the next line number and runs until that breakpoint is reached. For that reason, **next** runs much more quickly than **step**, and the difference is most noticeable when **step** is run from a line that calls a subroutine that has not been compiled with the debug flag. Those subroutines do not have line numbers and the debugger may have to run and stop at thousands of machine instructions until it reaches a point that corresponds to a line number. For that reason, you should use **next** on any line that contains a system call, and especially on those that involve input or output, such as **printf**.

There is no event number associated with these stops because there is no permanent event associated with stopping a program.

return [*procedure*]
: Continues execution until a return to *procedure* is encountered, or until the current procedure returns if *procedure* is not specified.

 If you accidentally step into a subroutine that you do not want to step through, you can use the **return** command to run through the current procedure or to a specified procedure.

skip [*num*]
: Continues execution until the end of the program or until *num* + 1 breakpoints execute.

Separating dbx Output from Program Output

This command is useful for debugging programs that are screen-oriented, such as text editors or graphics programs. If **screen** is not used, **dbx** output is intermixed with program output.

screen
: Opens a virtual terminal for **dbx** command interaction. The program continues to operate in the window in which it originated.

Tracing Execution

The **trace** command tells **dbx** to print information about the state of the program while the program is running. **trace** can slow your program considerably, depending on how much work **dbx** has to do. There are five basic forms of program tracing:

trace
Single steps the program, printing out each source line that is executed. This can be very slow for the same reasons that **step** can be slow.

trace in *procedure/function* [**if** *condition*]
Restricts the printing of source lines to when the specified *procedure* or *function* is active. You can specify an optional *condition* to control when trace information should be produced. For example:

```
(dbx) trace in sub2
[1] trace in sub2
(dbx) run
trace in hellosub.c:    8   printf("%s",s);
trace in hellosub.c:    9   i = '5';
trace in hellosub.c:   10   }
```

trace *procedure/function* **[in** *procedure/function***]** [**if** *condition*]
Displays a message each time *procedure* or *function* is called or returned from. When *procedure* is called, the information includes passed parameters and the name of the calling routine. On a return, the information includes the return value from *procedure*. The following is an example:

```
(dbx) trace sub
[1] trace sub
(dbx) run
calling sub(s = "hello", a = -1, k = delete) from function main
returning "hello" from sub
```

trace *linenumber* [**if** *condition*]
Prints the specified source line when the program reaches that line.

trace *expression* **at** *linenumber* [**if** *condition*]
Prints the value of *expression* when the program reaches the specified source line. The line number and file are printed, but the source line is not. For example:

```
(dbx) trace x*17 at "hellosub.c":8 if (x > 0)
[1] if x > 0 { trace x*17 } at "hellosub.c":8
(dbx) run
at line 8 in file "hellosub.c": x*17 = 51
```

trace *variable* **[in** *procedure/function***]** **[if** *condition***]**
Prints the location in the program and the value of *variable* each time the memory location associated with *variable* is modified. This is the slowest form of trace. The following is an example:

```
(dbx) trace x
[1] trace x
initially (at line 4 in "hello.c"):    x = 0
after line 17 in "hello.c":    x = 3
```

Deleting trace events is the same as deleting stop events. When a trace command is executed, the event ID associated is displayed along with the internal representation of the event.

Signal Handling

dbx can either trap or ignore signals before they are sent to your program. Each time your program is to receive a signal, **dbx** is notified. If the signal is to be ignored, it is passed to your program; otherwise, **dbx** stops the program and notifies you that a signal has been trapped. Change the default handling with the **catch** and **ignore** commands:

catch [*integer* | *signalname***]**

ignore [*integer* | *signalname***]**
Starts or stops trapping a signal before it is sent to your program. You can specify the signal by number or by name; names are not case sensitive, and the SIG prefix is optional. By default, all signals are trapped except **SIGCONT, SIGCLD, SIGALRM,** and **SIGKILL**. If no arguments are given, the defaults are listed.

In the following example, a program uses **SIGGRANT** and **SIGREQUEST** to handle allocation of resources. You do not want to stop **dbx** each time one of these signals is received:

```
(dbx) ignore GRANT
(dbx) ignore SIGREQUEST
%(dbx) ignore
CONT CLD ALARM KILL GRANT REQUEST
```

Calling Procedures

You can call any of your program procedures from **dbx** to test different arguments or you can call diagnostic routines that format data to aid in debugging. There are two ways to call a procedure:

call *procedure* **(***parm1, parm2, . . .* **)**
Runs the object code associated with *procedure*.

print *procedure* **(***parm1, parm2, . . .* **)**
Runs the object code associated with *procedure* and prints the value returned by *procedure*.

Displaying a Stack Trace

To get a listing of the procedure calls preceding a program halt, use the **where** command:

where [> *filename*]

In the following example, the executable object file `hello` consists of two source files and three procedures, including the standard procedure `main`. The program stopped at a breakpoint in procedure `sub2`.

```
(dbx) run
[1] stopped in sub2 at line 4 in file "hellosub.c"
(dbx) where
sub2(s = "hello", n = 52), line 4 in "hellosub.c"
sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
main(0x0, 0x0, 0x0), line 19 in "hello.c"
```

The stack trace shows the calls in reverse order. Starting at the bottom, the following events occurred:

1. Shell called `main` with three arguments, even though no arguments were declared in `main`. These three arguments represent the standard arguments passed into the main program: *argc, argv,* and *environ*.
2. `main` called procedure `sub` at line 19 with values `s = "hello"`, `a = -1`, and `k = delete`.
3. `sub` called procedure `sub2` at line 31 with values `s = "hello"` and `n = 52`.
4. The program stopped in procedure `sub2` at line 4.

 Note: Set the debugger variable **$noargs** to turn off the display of arguments passed to procedures.

You can also display portions of the stack with the **up** and **down** commands:

up [*count*]
down [*count*]
Move the current context in the stack by *count* levels.

For example:

```
(dbx) up 0
sub2(s = "hello", n = 54), line 4 in "hellosub.c"
(dbx) up 2
main(0x0, 0x0, 0x0), line 19 in "hello.c"
(dbx) down
sub(s = "hello", a = -1, k = delete), line 31 in "hello.c"
```

Refer to *Assembler Language Reference* for a description of the system calling conventions.

Printing Variables and Expressions

Use **dbx** to display a wide range of expressions. You specify expressions with a common subset of C and Pascal syntax, with some FORTRAN extensions.

***** (asterisk) or ^ (caret)	Denote indirection or pointer dereferencing.
[] (brackets) or **()** (parentheses)	Subscript array expressions.
. (period)	Use this field reference operator with pointers and structures. This makes the C operator → (arrow) unnecessary, although it is allowed.
& (ampersand)	Used to get the address of a variable.
.. (two periods)	Separates the upper and lower bounds when specifying a subsection of an array. For example: `n[1..4]`.

Displaying and Modifying Variables

print *expression* [, *expression* . . .]
Prints the values of the given expressions.
dump [*procedure*] [> *filename*]
Prints the names and values of the variables in *procedure*, or the current procedure if *procedure* is not specified.

assign *variable* = *expression*
Assigns the value of *expression* to *variable*.

To display an expression, use **print**. To print the names and values of variables, use **dump**. If the given procedure is . (period), then all active variables are printed. To modify the value of a variable, use **assign**.

For example, in a C program, you have an automatic integer variable `x` with value 7, and you are in procedure `sub2` with parameters `s` and `n`:

```
(dbx) print x, n
7 52
(dbx) assign x = 3*x
(dbx) print x
21
(dbx) dump
sub2(s = "hello", n = 52)
x = 21
```

Scoping of Names

Names resolve first using the static scope of the current function. The dynamic scope is used if the name is not defined in the first scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message **using** *qualified name* is printed. You can override the name resolution procedure by qualifying an identifier with a block name (such as *module.variable*). Source files are treated as modules named by the file name without the suffix. For example, variable `x`, which is declared in procedure `sub` inside file `hello.c`, has the fully qualified name `hello.sub.x`. The program itself has the name . (period).

Two **dbx** commands are helpful in determining which symbol is found when multiple symbols with the same name exist:

which *identifier*
Displays the full qualification of the given identifier (the outer blocks that contain the identifier).

whereis *identifier*
Displays the full qualification of all symbols that match the given identifier. Order is not meaningful.

The following is an example after stopping in `sub2`:

```
(dbx) which s
hellosub.sub2.s
(dbx) whereis s
hellosub.sub2.s hello.sub.s hello.main.s
```

The example shows there are three procedures in the program that have a symbol named `s`.

Operators Allowed in Expressions

The following types of operations are valid in expressions:

Algebraic	=, -, *,/(floating division), div (integral division), mod, exp (exponentiation)
Bitwise	-, \|, bitand, xor, ~, <<, >>
Logical	or, and, not, \|\|, &&
Comparison	<, >, <=, >=, <> or !=, = or ==
Other	sizeof

Logical and comparison expressions are allowed as conditions in **stop** and **trace** commands.

Type Checking in Expressions

Expression types are checked. You can override the type of an expression by using a renaming or casting operator. There are two forms of type renaming:

typename (expression)

expression \ typename

The following is an example where `x` is an integer with value 97:

```
(dbx) print x
97
(dbx) print char (x), x \ char, x
'a' 'a' 97
```

The **whatis** command prints the declaration of an identifier, which can then be qualified with block names as in the previous example.

whatis *name*

The following is an example:

```
(dbx) whatis sub2
int sub2(s,n)
char *s;
int n;
(dbx) whatis hello.sub.k
enum status k;
```

You can also print the declaration of an enumeration, structure, or union tag (or the equivalent in Pascal). The construct **$$***tagname* is used for that purpose:

```
(dbx) whatis $$status
enum $$status { run, create, delete, suspend };
```

The type of the **assign** command expression must match the variable type it is being assigned. If the types do not match, an error message is displayed. Change the expression type using a type renaming. You can disable type checking by setting a special **dbx** debugger variable named **$unsafeassign**. For example, using `n` and `status` as in the previous example:

```
(dbx) assign n = delete
incompatible types
(dbx) assign n = int (delete)
(dbx) print n, $$status (n)
2 delete
(dbx) set $unsafeassign
(dbx) assign n = suspend; print n
3
```

Folding Variables to Lower- and Upper-Case

By default, **dbx** folds symbols based on the current language. If the current language is C or undefined, the symbols are not folded; otherwise, the symbols are folded to lowercase. The current language is undefined if the program is in a section of code that has not been compiled with the debug flag. You can override default handling with the **case** command:

case [default | mixed | lower | upper]
Changes the way symbols are interpreted.

Using **case** without arguments tells you how case is currently being handled. In the following example, the current file is `testfort.f`, a FORTRAN program:

```
(dbx) case
Symbols are folded based upon current language
Symbols are currently folded to lower case
(dbx) print X
2
(dbx) case mixed
(dbx) print X
"X" is not defined
```

The FORTRAN and Pascal compilers fold all program symbols to lower case, whereas the C compiler does not.

Special Debugger Variables to Change Print Output

You can set the following special debugger variables to get different results from a **print** command:

$hexints	Prints integer expressions in hexadecimal.
$hexchars	Prints character expressions in hexadecimal.
$hexstrings	Prints the address of the character string, not the string itself.
$octints	Prints integer expressions in octal.
$expandunions	Prints fields within a union.

Set and unset these variables to get the desired results, for example:

```
(dbx) whatis x; whatis i; whatis s
int x;
char i;
char *s;
(dbx) print x, i, s
375 'c' "hello"
(dbx) print x, i, s
ox177 0x63 0x3fffe460
(dbx) unset $hexchars; set $octints
(dbx) print x, i
0567 'c'
```

You can set the following special debugger variables to get different debugging modes when using **dbx** in a windows environment:

- $source
- $machine

- $dual

See "Using dbx with X-Windows" on page 8-37 for more information on using **dbx** in a windows environment.

The variable **$catchbp** is available for use with the **next** command. Normally, execution of **next** does not honor breakpoints. In order to honor breakpoints with next set the special debugger variable **$catchbp**.

Displaying and Manipulating the Source File

You can use **dbx** to search through and display portions of the source files for a program. You do not need a current source listing. The **dbx** debugger keeps track of the current file, current procedure, and current line. If a core file exists, the current line and current file are set initially to the line and file containing the source statement where the process ended.

Note: This is only true if the process stopped in a location compiled for debugging. Otherwise, they are set to the first line in `main`. While you manipulate and list the source file, the values for current line and procedure may change.

Changing the Source Directory Path

The **dbx** debugger normally searches the current directory and the directory where the program is located for source files for the program. You can change this with the **-I** option on the **dbx** invocation line, or with the **use** command within **dbx**. For more information about the **-I** option, see "Other Invocation Options" on page 8-13.

use *directory1 directory2 . . .*
: Sets the list of directories to search for source files to the directories listed in this command. If no directories are listed, then the current source directory list is displayed.

Displaying the Current File

The **list** command allows you to list source lines:

list *linenumber, expression* [*, linenumber, expression*]
: Displays lines in the current source file from the first line number to the second line number, inclusive. If no lines are specified, the next 10 lines are listed unless the **$listwindow** debugger variable is set to some other value.

list *procedure|function*
: Displays lines surrounding the first line of *procedure*.

move *linenumber*
Change the next line listed to *linenumber*.

Some special symbols representing *source_line_numbers* are useful with the **list** command, and with the **stop** and **trace** commands. These symbols are **$** and **@**. The **$** symbol represents the next line to be run. The **@** symbol represents the next line to be listed. You can use simple integer expressions involving addition and subtraction in *source_line_number* expressions.
For example:

```
(dbx) list $
4 {
(dbx) list 5
5 char i = '4';
(dbx) list sub
23 char *sub(s,a,k)
24 int a;
25 enum status k;
 . . .
(dbx) move 25
(dbx) list @ -2
23 char *sub(s,a,k)
(
```

Changing the Current File or Procedure

You can use the following **func** and **file** commands to change the current file, current procedure, and current line within **dbx** without having to run any part of your program:

file [*filename*]
Changes the current file to *filename*. If no file is specified, the current source file name is displayed.
func [*procedure/function*]
Changes the current function. If no procedure or function is specified, then the current function is displayed. Changing the current function implicitly changes the current source file to the one that contains the function; consequently, it also changes the current scope used for name resolution.

You can search through the current file for text matching regular expressions. If a match is found, the current line is set to the line containing the matching text. The syntax of the search command is:

/ *regularexpression* **[/]**
Searches forward in the current source file for the given expression.

? *regularexpression* [?]
Searches backward in the current source file for the given expression.

If you repeat the search without arguments, the search wraps around the end of a file. For example:

```
(dbx) func; file
sub2 hellosub.c
(dbx) func sub
(dbx) func; file
sub hello.c
(dbx) / i
5 static int x;
(dbx) /
6 int i = 0xffffffff;
(dbx) ? static
5 static int x;
```

You can also invoke an external text editor for your source. The default editor invoked is **vi**. You can override this default by setting the environment variable **EDITOR** to your desired editor before invoking **dbx**.

edit [*filename*]
Invokes an editor on *filename*, or the current source file if *filename* is not specified.

edit *procedure/function*
Invokes an editor on the file containing the specified procedure or function.

Control returns to **dbx** after the return from an editing session.

Debugging Programs Involving Multiple Processes

Programs involving multiple processes are those that call the system routines **fork** and **exec**. When a program forks, the operating system creates another process that has the same image as the original; the original is called the ***parent*** process, and the created process is called the ***child*** process.

When a process performs an **exec** system call, a new program takes over or ***overlays*** the original process. Under normal circumstances, the debugger is only able to debug the original or parent process; however, **dbx** can follow the execution and debug the new processes.

You must issue the following command to enable multiprocess debugging:

multproc [on | off]
Enables or disables multiprocess debugging. The initial value is **off**. If you issue the command with no arguments, the current status is displayed.

When multiprocess debugging is enabled and a fork occurs, both the parent and the child process are halted in the fork. A separate virtual terminal is opened for a new version of **dbx** to control the running of the child:

```
(dbx) multproc
multi-process debugging is enabled
(dbx) run
```

When the fork occurs, execution is stopped in the parent, and **dbx** displays the state of the program:

```
application forked, child pid = 422, process stopped, awaiting input

trace trap in fork at 0x1000025a (fork+0xe)
(dbx)
```

Also, another virtual terminal is opened to debug the child:

```
debugging child, pid=422, process stopped, awaiting input

trace trap in fork at 0x10000250
10000250 (fork+0x4) )c05  jcs 1000025a (fork+0xe)
(dbx)
```

At this point there are two distinct debugging sessions. The debugging session for the child retains all the breakpoints from the parent process, but only the parent process can be rerun.

When a program performs an **exec** system call in multiprocess debugging mode, the program overwrites itself and the original symbol information becomes obsolete. All breakpoints are deleted when the **exec** occurs, and the new program is stopped and identified in order for the debugging to be meaningful. **dbx** attaches itself to the new program image, makes a system call to determine the name of the new program, reports the name, and then prompts for input:

```
(dbx) multproc
Multi-process debugging is enabled
(dbx) run
Attaching to program from exec . . .
Determining program name . . .
Successfully attached to /u/user/execprog . . .
Reading symbolic information . . .
(dbx)
```

Notes:

1. It can take some time for **dbx** to determine the name of the overlaying program.
2. It is not possible to determine the name of a program that belongs to a remote-mounted file system.

Debugging at the Machine Level

You can use **dbx** to examine programs at the assembler language level. You can display and modify memory addresses, display assembler instructions, single step instructions, set breakpoints and trace events at memory addresses, and display the registers.

In the commands and examples that follow, an ***address*** is an expression that evaluates to a memory address. The most common forms of addresses are integers and expressions that involve taking the address of an identifier with the & (ampersand) operator. You can also specify an address as an expression enclosed in parentheses in machine level commands. Addresses can be composed of other addresses and the operators + (plus), - (minus) and indirection (unary *).

Machine Registers

Use the **registers** command to see the values of the machine registers.

registers [> *filename*]
Displays the values of the machine registers and the current instruction.

Registers are divided into three groups:

- General-purpose registers are denoted by **$r***n*, where *n* represents the number of the register.
- Floating-point registers are denoted by **$fr***n*, where *n* represents the number of the register.
- Supported system-control registers are:

- The instruction-address register, **$iar** or **$pc**
- The condition-status register, **$cs**
- The interrupt-control status register, **$ics**
- The multiplier-quotient register, **$mq**.

Examining Memory Addresses

Use the following to print the contents of memory starting at the first address and continuing up to the second address, or until *count* items are displayed. The mode specifies how memory is to be printed.

address, address / [*mode*] [> *filename*]
address / [*count*] [*mode*] [> *filename*]

If *mode* is omitted, the previous mode specified is reused. The initial mode is **X**. The following modes are supported:

b	Prints a byte in octal.
c	Prints a byte as a character.
d	Prints a short word in decimal.
D	Prints a long word in decimal.
f	Prints a single-precision floating point number.
g	Prints a double-precision floating point number.
h	Prints a byte in hexadecimal.
i	Prints the machine instruction.
o	Prints a short word in octal.
O	Prints a long word in octal.
s	Prints a string of characters terminated by a null byte.
x	Prints a short word in hexadecimal.
X	Prints a long word in hexadecimal.

Note the following example:

```
(dbx) print &x
0x3fffe460
3fffe460: 31323300
(dbx) &x,&x+12/x
3fffe460: 3132 3300 7879 7a5a 5958 5756 003d 0032
(dbx) ($pc)/2i
100002ce (sub)   c8000034 cal r0,34
100002d2 (sub+0x4) 1001    stcs r0,0(r1)
```

Expressions in parenthesis can be used as an address, as in the preceding example.

Running Your Program at the Machine Level

The commands for debugging your program at the machine level are similar to those at the symbolic level. The following two commands stop the machine when the address is reached, or the condition is true, or the variable is changed:

stopi [*variable***]** [**if** *address*] [**if** *condition*]

stopi *address* [**if** *condition*]

The following commands are similar to the symbolic trace commands:

tracei [*variable***]** [**at** *address*] [**if** *condition*]
: Prints the location of the machine instruction and value of *variable* each time memory *address* is accessed, *variable* is changed, or *condition* is true.

tracei *address* [**if** *condition*]
: Prints the specified address whenever the instruction at that address is executed.

stepi [*num*]
: Executes one or *num* machine instruction(s).

nexti [*num*]
: Executes up to the next or next *num* machine instruction(s). The **nexti** subcommand does not follow branch-and-link instructions; it continues until execution returns to the next instruction.

gotoi *address*
: Changes the program counter to address.

Other commands such as **assign** and **return** work at the machine level. In fact, if there are no symbolic addresses applicable, **stop, trace, step,** and **next** can be used equivalently to the machine-level counterparts.

When your program stops in a procedure that has not been compiled for debugging, the next instruction to be executed is displayed along with the current machine address. This is analogous to the current line at the symbolic level. For example:

```
(dbx) stopi at &sub
[1] stopi at 0x100002d0 (sub)
(dbx) run
[1] stopped in sub at 0x100002d0
100002d0 (sub)  c8000034  cal  r0,34
(dbx) step
stopped in sub at 0x100002d4
100002d4 (sub+0x4) 8df00042 balix r15,1000035c (printf)
```

If you performed another **stepi** at this point, you would stop at address 0x1000035c, identified as the entry point of procedure `printf`. If you do not intend to do that, you

could **return** to continue execution at the next instruction in `sub` at address 0x100002e0. On the other hand, **nexti** at this point continues execution to 0x100002e0 automatically.

The dbx Debugging Environment

The following topics show how to modify and change the **dbx** environment.

The Alias Facility

You can build your own commands from the **dbx** primitive command set. The following commands allow you to build a user alias from the arguments specified. All commands in the replacement string for the alias must be **dbx** primitive commands. You can then use your aliases in place of the **dbx** primitives.

alias [*aliasname* [*commandname*]]

alias *aliasname "command string"*

alias *aliasname* **(***parm1, parm2, . . .* **)** *"command string"*

The **alias** command with no arguments displays the current aliases in effect; with one argument the command displays the replacement string associated with that alias.

The first two forms of **alias** are used to simply substitute the replacement string for the **alias** each time it is used. For example:

```
(dbx) alias rr rerun
(dbx) alias step2 "step;step"
```

Each time `rr` is typed at the command line, **dbx** performs a **rerun** command. Similarly, **step2** results in two **step** commands being executed.

The third form of aliasing is a limited macro facility. Each parameter specified in the **alias** command is substituted for in the replacement string. This can be useful in eliminating excessive typing:

```
(dbx) alias px(n) "set $hexints; print n; unset $hexints"
(dbx) alias a(x,y) "print symname[x]→symvalue._n_n.name.Id[y]"
(dbx) px(126)
0xfe
```

The alias `px` in the previous example prints a value in hexadecimal without permanently affecting the debugging environment. The following aliases and associated command names are supplied by **dbx** by default:

t	**where**	**p**	**print**
j	**status**	**n**	**next**
st	**stop**	**l**	**list**
s	**step**	**e**	**edit**
r	**run**	**h**	**help**
x	**registers**	**d**	**delete**
q	**quit**	**c**	**cont**

You can remove an alias with the **unalias** command:

unalias *aliasname*
Removes an alias with the given name.

Changing the dbx Prompt

The **dbx** prompt is normally the name with which you invoked **dbx**. If you specified `/usr/bin/dbx a.out` on the command line, then your prompt would be `/usr/bin/dbx`. You can change the prompt to be whatever you desire with the **prompt** command:

prompt *"promptstring"*
Changes the **dbx** prompt to *promptstring*.

For example:

```
$ /usr/bin/dbx hello
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information . . .
(/usr/bin/dbx) prompt "dbg>"
dbg>
```

Customizing Your Environment with .dbxinit

Each time you begin a debugging session, **dbx** searches for a special initialization file named **.dbxinit**. This file is searched for first in the current directory, and then in your home directory. **.dbxinit** should contain a list of **dbx** commands to execute each time you begin a debugging session. These commands are executed before **dbx** begins to read commands from standard input. Normally, **.dbxinit** contains **alias** commands, but it can contain any valid **dbx** commands. For example:

```
$ cat .dbxinit
alias si "stop in"
prompt "dbg>"
$ dbx a.out
dbx version 2.2 for AIX
Type 'help' for help.
reading symbolic information . . .
dbg> alias
si      stop in
t       where
 . . .
dbg>
```

Reading dbx Commands from a File

The **-c** invocation option and `.dbxinit` provide mechanisms for executing **dbx** commands before reading from standard input. There is also a way to read **dbx** commands from a file once the debugging session has begun:

source *filename*
: Reads **dbx** commands from the given file name.

Running Shell Commands from dbx

You can run shell commands without exiting from the debugger:

sh *commandline*
: Passes *commandline* to the shell for execution.

If *commandline* is missing, the shell is entered for use until it is exited, at which time control returns to **dbx**. The **SHELL** environment variable determines which shell is used. For example:

```
(dbx) sh echo $SHELL
/bin/sh
(dbx) sh
$ echo 'This is the shell' #You will remain in the shell until you exit.
$ exit
(dbx)
```

Getting Help

help Prints a synopsis of common **dbx** commands.

sdb to dbx Equivalencies

The following are equivalent functions in **sdb** and **dbx** with a short explanation of each:

sdb	**dbx**	**Description**
*proc*b	stop in *proc*	Stops in procedure *proc*.
B	status	Lists breakpoints.
#b	stop at #	Stops at line #.
s	step	Steps one source line.
S	next	Steps to next source line.
x	registers	Displays registers.
q	quit	Exits Debugger.
i	stepi	Steps one machine instruction.
var/	print *var*	Displays the value of *var*.
var!*value*	assign *var* = *value*	Modifies value of the variable.
t	where	Displays stack trace.
!	sh	Escapes to shell.
var=	print &*var*	Prints address of variable.
c	cont	Continues execution.
T	up 0	Prints top line of stack trace.
p	list $	Lists the next line to be executed.
w	list $-5,$+5	Lists 10 lines around the current line.
z	list @,@+10	Lists 10 lines from the current line.
*var*m	stop *var*	Stops when the variable changes.
*proc*a	trace *proc*	Traces calls to *proc*.
D	delete all	Remove all breakpoints.

Using dbx with X-Windows

If you have X-Windows installed on your system, you can use the **xdbx** command to run the **dbx** symbolic debugger in a multiwindow environment. There are no command options specifically for **xdbx**; options pass directly to **dbx**.

To start **xdbx** from the X-Windows display, type `xdbx` and the **dbx** options you want. The format for the start **xdbx** command is: **xdbx** *dbx_command_options filename*. You can run the display portion of **xdbx** on a remote station by entering: **-x** *option*: (*optional display*) . Where *option* is the name of the remote host, and the *optional display* is a small integer designating the display. The default is 0.

Notes:

1. You must start X-Windows before using the **xdbx** command. To start X-Windows, type `xinit` at the prompt.
2. You can run **xdbx** with AIX X-Windows Release 1.1 or later.

The xdbx Window Environment

Windows in **xdbx** contain menus, provide information, or both. The three menus are the source, instruction, and format menus. These are pop-up menus, and appear only when selected with the right mouse button. You also select items in a menu window with a mouse. For a discussion on using a mouse see "Using the Mouse" on page 8-41.

Windows specific to the **xdbx** mode are automatically displayed at the start of a session. The status and command windows are also displayed automatically and appear in all **xdbx** modes. These windows remain displayed throughout an **xdbx** session.

Other windows appear only when you issue a specific command or when the system generates informational and error messages. The watch window displays data requested with a **watch** command. The message window displays messages generated by **xdbx** or **dbx**.

The Command Window

This window interacts with **dbx**. It is the input/output window for the debugger. You can type commands on the command line or select a command from a window menu. Commands selected from a menu are echoed in the command window. These commands are displayed on the last line of the command window.

Retrieve previously entered commands with the **Cursor Up** or **Cursor Down** key. You can edit and send commands from the command window to **dbx**.

Note: The command line is 256 characters long (including the prompt) and cannot be continued to the next line.

To change the prompt, type: `prompt` *newprompt*, and press **Enter**. To clear the command line, press and hold the **CTRL** key and press the **u** key.

The Format Menu

The Format menu contains attributes applied to address values selected from the instruction and register windows. The *nn* value in the menu title line indicates the number of information units to display. Available units selected at the Format menu include:

- String
- Octal byte
- ASCII byte
- Short or long dec
- Short or long hex
- Short or long octal
- Single or double float

Change the value of *nn* by moving the mouse pointer horizontally across the selected menu line. The leftmost position on the menu line represents the value 1 and the rightmost represents the value 35. This window is available in the **$machine** or **$dual** mode.

The Instruction Window

The instruction window displays assembler instructions around the address where the debugger is stopped. Markers indicate the current line and the location of breakpoints:

- A stop sign indicates a breakpoint.
- A yield sign indicates a conditional breakpoint.
- An arrow indicates a current line.

This window is available only in the machine or dual mode.

The Instruction Menu

The instruction menu contains machine level **dbx** debugging commands. This pop-up menu is available only with the instruction window.

The Message Window

The message window displays error and informational messages generated by **dbx** and **xdbx**. This window is automatically generated when an error is detected. Messages display in the order received. If the window is already active, additional messages are added to the bottom of the window. The size of the window increases with the number of messages received. Remove the window by selecting the remove button in the left corner of the window.

The Register Window

The register window displays the current value of each hardware register. The window is updated when **dbx** stops at a breakpoint. A scroll bar displays with the register window. This window is available in machine or dual mode.

The Source Window

The source window displays the source code around the line Displays the source code around the line where the debugger is stopped. You can also display other parts of the source file or source code from other files. The source window is available in the **$source** or **$dual** mode.

Markers show the current line and the location of breakpoints:

- A stop sign indicates a breakpoint.
- A yield sign indicates a conditional breakpoint.
- An arrow indicates the current line.

A scroll bar is present on the left side of the source window. The entire scroll bar represents the total source file. The shaded area represents the location and relative size of the displayed lines.

The Source Menu

The source menu window contains source level **dbx** debugging commands. This pop-up menu is available only with the source window.

The Status Window

The status window displays the current status of the program being debugged:

- Executing
- Tracing
- Stopped in
- Execution complete.

The status window also displays the file name of the current file and the **xdbx** process identifier.

The Watch Window

The watch window displays data requested in a **watch** command.

xdbx Modes

The **xdbx** debugger displays up to three separate menu windows in a deck-of-cards style depending on the debugging mode. You can select one of three modes:

- Source
- Machine
- Dual.

You specify a debugging mode by entering one of the following:

set $source The default mode if *any* source code compiled with the **-g** flag is available. Only the source window is displayed. The source menu window contains source level **dbx** debugging commands.

set $machine The default mode if no code compiled with **-g** is available. Displays the instruction and register windows. The instruction menu window contains machine-level **dbx** debugging commands.

Note: Select attributes of the memory locations displayed in the instruction and register windows from the Format menu. For information on the attributes available, and the method of selection see "The Format Menu" in this chapter.

set $dual Displays the source, instruction, and register windows.

Using the Mouse

This section describes how to use the mouse to select items and initiate actions in the various **xdbx** windows.

In a Window

Use the left mouse button to select one or more items in a window; then select the action to be performed from the pop-up menu.

To deactivate an item, select it a second time. If a selected item is scrolled off the screen before an action is taken, the item is deactivated.

In the Scroll Bar

Press and release the left mouse button to display source lines toward the end of the source file.

Press and release the right button to display source lines toward the beginning of the source file.

Simultaneously press and release both buttons to display source lines relative to the position of the pointer in the scroll bar window.

In a Menu Window

When you press the right mouse button, the pop-up menus for an **xdbx** mode appear. The menu pointer is positioned on the last command selected. Hold down the right button and move the pointer to the desired command. Release the button to select the highlighted command and remove the menu window.

If you do not want to make a selection from a menu, move the pointer outside the menu window and release the button.

A separate **dbx** command is generated for each item selected in the source and command windows. For example, if you select two variables in the source window and one variable in the command window, and then `print` is selected in a menu window, three separate **print** commands are sent to **dbx**.

A menu command can have one of the following five interpretations as specified in the **.Xdefaults** file (see "Changing the xdbx Environment" on page 8-42):

none The command does not require a selected item (such as **step**).

address The address of the selected assembly instruction is appended to the menu command (such as **stopi at**).

line The line number of the line containing the selected item is appended to the menu command (such as **stop at**).

text The selected area is appended to the menu command (such as **print**).

If a single character is selected, the selection is expanded to include all alphanumeric characters, underscores, and dollar signs (a-z, A-Z, 0-9, _, and $). If more than one character is selected (you moved the mouse while holding down the left button), no automatic expansion occurs. In this case, you get what you select. In either case, the selected area does not span display lines.

command The complete line is sent to **dbx** in place of the selected menu command (such as **again**).

Any menu command that requires selected items (such as **print**) acts on, and then deactivates, all the selected items in both the source and command windows. These menu commands generate an informational error message if no items have been selected in the source, command, register, or instruction window.

Menu commands that do not require selected items (such as **step**) do not remove previously selected items in the source, command, register or instruction window.

Changing the xdbx Environment

You can change the **xdbx** environment by changing or adding entries in the **.Xdefaults** file. You can change:

- The font for each window
- The initial position of each window (except the message window)
- The number of lines in the source, command, instruction, and watch windows
- The menu commands
- The foreground and background colors.

The **.Xdefaults** file must be in the user's **$HOME** directory. The following is a list of the user controllable display parameters; default values are shown if provided:

```
XDBX.statusfont:        Rom14.500
XDBX.statusgeom         =95x2+0+0
XDBX.statusfgcolor:     black
```

```
XDBX.statusbgcolor:     white
XDBX.sourcefont:        Rom14.500
XDBX.sourcegeom         =WxH+X+Y
XDBX.sourcefgcolor:     black
XDBX.sourcebgcolor:     white
XDBX.commandfont:       Rom14.500
XDBX.commandgeom        =WxH+X+Y
XDBX.commandfgcolor:    black
XDBX.commandbgcolor:    white
XDBX.commandlines       25
XDBX.watchfont:         Rom14.500
XDBX.watchgeom          =WxH+X+Y
XDBX.watchfgcolor:      black
XDBX.watchbgcolor:      white
XDBX.instfont:          Rom14.500
XDBX.instgeom:          =WxH+X+Y
XDBX.instfgcolor:       black
XDBX.instbgcolor:       white
XDBX.registerfont:      Rom14.500
XDBX.registergeom       =WxH+X+Y
XDBX.registerfgcolor:   black
XDBX.registerbgcolor:   white
XDBX.menufont:          Rom10.500
XDBX.menufgcolor:       black
XDBX.menubgcolor:       white
XDBX.menudispname:      SRC MENU
XDBX.menu2dispname:     MACHINE MENU
XDBX.messagefont:       Rom10.500
XDBX.messagefgcolor:    black
XDBX.messagebgcolor:    white
XDBX.scrollfgcolor:     black
XDBX.scrollbgcolor:     white
XDBX.stopsigncolor:     black
XDBX.yieldsigncolor:    black
XDBX.arrowcolor:        black
XDBX.menucommand1:      stop at, line
XDBX.menucommand2:      stop in, text
XDBX.menucommand3:      print, text
XDBX.menucommand4:      print *, text
XDBX.menucommand5:      listi at, line
XDBX.menucommand6:      rerun, none
```

```
XDBX.menucommand7:      step, none
XDBX.menucommand8:      next, none
XDBX.menucommand9:      cont, none
XDBX.menucommand10:     again, command
XDBX.menucommand11:     clear, line
XDBX.menucommand12:     quit, none
XDBX.menu2command1:     stopi at, address
XDBX.menu2command2:     registers, none
XDBX.menu2command3:     rerun, none
XDBX.menu2command4:     stepi, none
XDBX.menu2command5:     nexti, none
XDBX.menu2command6:     cont, none
XDBX.menu2command7:     again, command
XDBX.menu2command8:     cleari, address
XDBX.menu2command9:     quit, none
```

Changing or Creating Menu Commands

You can create new menu commands by simply adding them to the *.Xdefaults* file; the commands are numbered 1 through nn (xdbx.menucommand[nn]). No checking is done on user-supplied **dbx** commands. Use a , (comma) as the delimiter between the command and the selection interpretation. The following example changes command line 6 and adds 2 new commands:

```
xdbx.menucommand6:      run,none
xdbx.menucommand12:     status
xdbx.menucommand13:     whereis,text
```

The first 11 commands can be in any order. Command numbers greater than 11 must be numerically contiguous and in numerical order. For example, command13 cannot be added without adding command12, and command12 must precede command13.

Changing the Geometry Specification

You can position the status, source, command, and watch windows individually because they are not relative to one another. In the position specification:

- W is specified in characters.
- H is specified in lines.
- X and Y are specified in pixels.

Attaching a Process

To attach a process that is already running issue the following commad:

xdbx -a*pid*

Where `pid` is the process ID of the running process to be debugged. **dbx** determines the path name of the program and attaches to it. An error message is issued and debugging terminated if:

- You specify an invalid or no pid.
- The path name of the program cannot be determined.
- You do not have authority to debug the program.

When you get control, it is similar to other **dbx** sessions with the following exceptions:

- The program is not stopped at the beginning of the program but rather at some point within the program.
- **run** and **rerun** commands are disabled. The reason for this is that both **run** and **rerun** reload the program as a child of **dbx**, and **dbx** has no knowledge of the original parameters passed to the program.

All other **dbx** commands can be used just as in a normal debug session. To terminate an attached debug session, use one of the following two options:

quit Terminates the program, **dbx**, and **xdbx**.

detach
Removes all breakpoints from the application, and the program and both **dbx** and **xdbx** terminate. However, the application program continues to run.

Debugging Multiprocess Programs

Use the **multproc [on/off]** command to turn on or off the multiprocess debugging function. **dbx** comes up in the off state.

A new copy of **xdbx** is generated whenever a new process is started and **dbx** is enabled for multiprocessing.

If the Process Issues a fork Command

When an application being debugged issues a **fork** command, a new **xdbx** window appears that reflects the child of the application being debugged. A message appears indicating the child application has been created. The child process is in trace mode similar to a normal debug session. You can enter the **dbx** command **step**, **next**, or **cont** to start processing.

run and **rerun** commands are disabled for the child process.

The parent process and its debugger stops and prints a message similar to when a breakpoint occurs. To continue debugging the parent, you can use the **step**, **next**, or **cont** command.

If the Process Issues an exec Command

If an application being debugged issues an **exec** command, a message appears indicating an **exec** has taken place. The new program is in trace mode similar to a normal debug session, and you can enter **dbx** commands at this point. Use either the **step**, **next**, or **cont** command to start processing.

run and **rerun** commands are disabled because these commands reload the program as a child of **dbx**; **dbx** has no knowledge of the original parameters passed to the program.

The sdb Program

The **sdb** program performs three functions:

- Examines core images of programs that did not end properly.
- Displays and manipulates sections of source text.
- Controls and monitors the execution of programs.

The **sdb** commands that perform these functions are usually a single letter or character. Using **sdb**, you can:

- Debug programs at either the source language or assembler language level.
- Issue commands from either standard input or a named file.
- Access variables, including arrays and structure elements, symbolically and display them in the correct format.
- Display or modify the contents of machine registers.
- Examine the source text using simple search and scrolling functions.
- Set breakpoints at selected statements.
- Run a program one line at a time.
- Call program or diagnostic procedures directly from the debugger.

Refer to *AIX Operating System Commands Reference* for complete reference information about the **sdb** program.

sdb Command Summary

Start/Stop sdb

sdb <*filename*>
: Starts **sdb** using file <*filename*>

e <*name*>
: Uses the file or procedure indicated by <*name*>

e
: Displays the name of the current file and current procedure

q
: Quits **sdb**

Setting Breakpoints

[*procedure*][*line number*]**b**
: Sets a breakpoint at *line number* in *procedure.*

B
: Displays a list of all breakpoints that are set.

[*procedure*][*line number*]**d**
: Removes a breakpoint at *line number* in *procedure.*

D
: Removes all breakpoints.

[*variable*]**$m**[*count*]
: Runs current procedure one line at a time until *variable* changes or *count* number of lines are run.

[*address*]**:m**[*count*]
: Runs current procedure one line at a time until *address* changes or *count* number of lines are run.

Setting Trace Options

[*procedure*]**a**
: Displays procedure name and its arguments each time it is called.

[*procedure*][*line number*]**a**
: Displays source line *line number* each time it is executed.

t
: Traces procedure calls.

Running the Program from sdb

r *args*
: Runs the current procedure using *args* as the starting parameters.

r
: Runs the current procedure using the same starting parameters as the last time the procedure ran.

R
: Runs the current procedure with no starting parameters.

[*procedure***][***line number***]c**
: Continues running a stopped program.

[*procedure***][***line number***]C**
: Continues running a stopped program; passes signal that stopped program back to program (test signal handler).

[*line number***]g**
: Starts running the program at *line number* in the current procedure.

k
: Stops the program being debugged.

Single Stepping

[*level***]v**
: Sets level of information displayed during single step mode.

s
: Runs a single line of source code.

S
: Runs a single line of source code, but does not stop during execution of called procedures.

i
: Runs a single assembler instruction rather than source line; ignores signals that stop the program.

I
: Runs a single assembler instruction rather than source line; passes signals that stop the program to the program.

Displaying Information

variable/
: Displays value of *variable*.

[*procedure***]:[***line number***]?**
: Displays assembler language statements associated with *line number*.

[*address***]:?**
: Displays assembler language statement associated with *address* in text space.

x
: Displays the values of all registers and the process status word.

X
: Displays the value of IAR and the assembler language instruction at that location.

variable=
: Displays address of *variable*.

p
: Displays current line.

w
: Displays 10 lines around the current line.

z
: Displays 10 lines starting at the current line. Changes current line to current line + 10.

ctrl-d
: Displays the next 10 lines of instructions, source code or data.

Changing Values of Program Variables

variable!*value*
: Assigns new *value* to *variable*.

Changing Position in the File

n
: Sets current line to *n*.

Enter key
: Moves current line forward 1 line.

+[*n***]**
: Moves current line forward *n* lines (1 line if no *n*).

-[*n***]**
: Moves current line backward *n* lines (1 line if no *n*).

/string
: Searches forward for *string*.

?*string*
Searches backward for *string*.

Reference Metacharacters

*	Matches any sequence of characters.
?	Matches any single character.
.	Refers to the last referenced variable.

Using the Program

To use **sdb** with a program, compile the source program with the **-g** flag. The compiler generates additional information about the variables and statements of the compiled program.

The following rules apply to the use of the **-g** flag:

- The **sdb** program displays an error message if used on a main program compiled without the **-g** flag.
- Use **sdb** for any procedures compiled with the **-g** flag, even if the main program was compiled without the **-g** flag.
- If an error occurs in a procedure compiled without the **-g** flag, **sdb** displays only the procedure name and the address at which the error occurred.

The following example shows the generation of an executable program called `samp`. When the program runs, it produces a ***bus error*** and stops. The commands that follow the error message show how to start and end the **sdb** program:

```
$ cc -g samp.c -o samp
$ samp
Bus error - core dumped
$ sdb samp
main:25:    x[i] = 0;
*q
$
```

In the example, the program `samp` has an error which causes a core dump. When **sdb** starts, it reports that the bus error occurred in procedure `main` at line 25 (line numbers are always relative to the beginning of the file) and then displays line 25. The * indicates that **sdb** is waiting for an **sdb** command. The **q** command then exits the **sdb** program.

In the preceding example, **sdb** is called with one argument, `samp`. The **sdb** program can have up to three arguments on the command line:

- The name of the executable file to be debugged. Without a file name, **sdb** uses **a.out**.
- The name of the core file. Without a file name, **sdb** uses **core**.
- The name of the directory containing the program source. The **sdb** program requires all source to reside in a single directory. Without a file name, **sdb** uses the current directory.

In the example, the second and third arguments used the correct values, so only the first was specified.

You can redirect standard input or output while in **sdb**. Therefore, **sdb** command input can come from a file and output can be sent to another file or even the printer. Redirect input and output with the < and > commands. When redirecting standard input, all **sdb** commands come from a named file until the end of the file is reached. The input then reverts to the keyboard. This command cannot be nested. The following example redirects input to come from a file of **sdb** commands called `sdbbatch`:

```
*< sdbbatch      # input now comes from sdbbatch
```

Displaying a Stack Trace

To get a listing of the procedure calls that led to an error, use the **t** command. In the following example, the executable file `samp.c` contains the standard procedure `main` and two additional procedures called `last` and `middle`:

```
*t
last(x=2,y=3)        [samp.c:25]
middle(i=16012)        [samp.c:96]
main(argc=1,argv=0x7fffff54,envp=0x7fffff5c)      [samp.c:15]
```

The stack trace shows the calls in reverse order. Starting at the bottom, the following events occurred:

1. **shell** called `main` with three arguments referred to as `argc`, `argv` and `envp`. Note that `argv` and `envp` are pointers, so their values are written in hexadecimal.
2. `main` called `middle` at line 15 with the value `i=16012`.
3. `middle` called `last` at line 96 with the values `x=2, y=3`.
4. `last` contained an error at line 25.

Refer to *Assembler Language Reference* for a description of how the system calling conventions operate.

Examining Variables

To display the value of a variable, type the variable name followed by a slash:

```
*countvar/      # display the value of countvar
```

Unless otherwise specified, variables are assumed to be either local to or useable by the current procedure. To specify a different procedure, use the form:

```
procedure:variable/
```

The **sdb** program supports a limited form of pattern matching for variable and procedure names. The symbol * is used to match any sequence of characters of a variable name and ? to match any single character. Consider the following commands:

```
*x*/
*procwild:y?/
**/
```

The first writes the values of all variables beginning with `x`, the second writes the values of all two letter variables in procedure `procwild` beginning with `y`, and the third writes all variables. In the first and third examples, only variables useable by the current procedure are written.

The following command displays the variables for each procedure on the call stack:

```
**:*/
```

To reference a particular instance of a variable on the stack, use the form:

```
procedure:variable,number
```

For example, the following command displays the value of the variable y in the call to `procwild` closest to the top of the stack:

```
*procwild:y,1/
```

To match only global variables, use the form:

```
:variable
```

The **sdb** program usually displays variables in a format determined by their type as declared in the source program. To request a different format, use the form:

```
variable/[count][length][format]
```

Without a *format* parameter, **sdb** uses decimal notation.

This command is explained in detail in the reference information for **sdb** (see *AIX Operating System Commands Reference*).

For example, the variable `i` can be displayed in hexadecimal with:

```
*i/x
```

To specify `i` in short form in hexadecimal, use the *length* specifier `h`, along with the *format* specifier `x`:

```
*i/hx
```

Memory locations can also be displayed by specifying their absolute addresses. The following command displays location 1024 in decimal:

```
*1024/
```

Because numbers can be specified in octal or hexadecimal, the following two commands also display location 1024 in decimal:

```
*02000/
*0x400/
```

You can also mix numbers and variables. The next example refers to an element of a structure starting at address 1000:

```
*1000.x/
```

The following example refers to a structure element whose address is at location 1000:

```
*1000->x/
```

For commands of the type `*1000.x/` and `*1000->x/`, **sdb** uses the format of the last structure referenced.

The address of a variable is displayed with the `=` command:

```
*i=
```

The dot command redisplays the last referenced variable:

```
*./
```

Referencing Arrays, Pointers, and Structures

The **sdb** program also works with arrays, pointers, and structures, such as:

```
*array[2][3]/
*sym.id/
*psym->usage/
*xsym[20].p->usage/
```

The only restriction is that array subscripts must be numbers. Note that as a special case:

```
*psym->/d
```

displays the location pointed to by `psym` in decimal.

Elements of a multidimensional array may be referenced as variable[number][number]..., or as variable[number,number...]. *Number* can be one of the following:

number;number Indicates a range of values.

***** Indicates all legal values for that subscript.

omitted Does not use a *number* parameter to display the full range of values if the number is the last subscript.

As with structures, **sdb** normally displays all the values of an array or of the section of an array if trailing subscripts are omitted. When the = command is applied to an array, **sdb** displays only the address of the array itself or of the section specified by the user if subscripts are omitted. The following are all valid references to the two-dimensional array a[5][6]:

```
*a/
*a[2;5][4]/
*a[3]/
*a[3,4]/
*a[*][3]/
```

Displaying and Manipulating the Source File

The **sdb** program allows you to search through and display portions of the source files for a program. You do not need a current source listing. The **sdb** program keeps track of the current file, current procedure and current line. If the core file exists, the current line and current file are initially set to the line and file containing the source statement where the process ended. Otherwise, they are initially set to the first line in `main`. While

manipulating the source file, the values for current line and current procedure may change.

Displaying the Current File

Four commands exist for displaying lines in the current file:

p	*Print* displays the current line.
w	*Window* displays a window of 10 lines around the current line.
z	Displays 10 lines starting at the current line. Advances the current line by 10.
ctrl-d	*Scroll* displays the next 10 lines and advances the current line by 10.

The **sdb** program displays the line number corresponding to the displayed line. The line number not only indicates the relative position of the line in the file, but some **sdb** commands also use it as input.

Changing the Current File or Procedure

Use the **e** command to change the current file or procedure. In both cases, the current line becomes the first line in the procedure or file. Use one of the following two forms:

```
*e file.c
*e procedure
```

An **e** command without an argument displays the name of the current file and current procedure.

Changing the Current Line

The **z** and **ctrl-d** commands change the current line in the source file. The following commands also change the current line:

/string/	Searches forward for *string*. You can omit the second /.
?*string***?**	Searches backward for *string*. You can omit the second ?.
+[*n***]**	Moves the current line forward *n* lines.
-[*n***]**	Moves the current line backward *n* lines.
enter key	Moves the current line forward one line.

n Sets the current line to *n*.

You can combine these commands with the display commands. The following example advances the current line by 15, displays 10 lines, and advances the current line by 10:

```
*+15z
```

Controlling Program Execution

The **sdb** program allows you to set ***breakpoints***, or stopping places in the program. After entering **sdb**, specify what lines in the source program are to be breakpoints. Then run the program from within **sdb**. When the program reaches a breakpoint, it stops running. The **sdb** program reports that it has reached a breakpoint. Then use **sdb** commands to examine the procedure calls and variables. If the program is working correctly to this point, you can add or remove other breakpoints and continue running the program.

An alternative to setting breakpoints is single stepping, which causes the program to run a single line of code and then stop. If the procedure was not compiled with the **-g** flag, the program runs until **sdb** finds a procedure that was compiled with the **-g** flag. You can also execute one assembler instruction at a time.

Setting and Deleting Breakpoints

Breakpoints can be set at any line in a procedure that contains executable code. Enter the command as follows:

```
[procedure][line number]b
```

The following examples show how to set a breakpoint:

```
*12b            # at line 12 in current procedure
*samp:12b       # at line 12 in procedure samp
*samp:b         # at the first line of samp
*b              # at the current line
```

The line numbers are relative to the beginning of the file as written by the source file display commands.

Remove breakpoints using the letter **d** instead of **b**. The only difference is when using **d** by itself, **sdb** displays each breakpoint and asks whether to remove it or not. If you respond with **y** or **d**, **sdb** removes the breakpoint.

To display a list of the current breakpoints, use the **B** command. To remove all of the breakpoints, use the **D** command. Remember that **sdb** commands are case sensitive. There is a big difference between the **d** command and the **D** command.

The **b** command can also automatically perform a sequence of commands at a breakpoint and then continue running the program. For example, the command:

```
*12b t;x/
```

displays both a trace back and the value of `x` each time the program runs line 12.

The **a** command is a variation of the preceding command. There are two ways of using the **a** command:

```
*samp: a
*samp:12 a
```

The first displays the procedure name, here `samp`, and its arguments each time it is called. The second displays the source line, 12, each time it is about to be executed. For both forms of the **a** command, execution continues after the procedure name or source line is displayed.

To change the value of a variable when the program is stopped at a breakpoint, use the following command format:

```
variable!value
```

The value can be a number, character constant, register, or the name of another variable. If the data type of the variable is float or double, the value can also be a floating-point constant.

Running the Program

The **r** command starts running the program. The command format is:

`r` args

The program runs using the arguments as if they were typed on the **shell** command line. If arguments are not specified, then the arguments from the last execution of the program are used. To run a program without arguments, use the **R** command.

After the program starts, it runs until one of the following occurs:

- It reaches a breakpoint
- A signal, such as **INTERRUPT** or **QUIT**, occurs
- The program ends.

In each case when the program stops, **sdb** receives control and displays a message to tell why the program stopped.

Use the **c** command to continue running a stopped program. To specify a line number in the **c** command:

```
*procwild:12 c
```

places a temporary breakpoint at the named line. The breakpoint is removed when the **c** command finishes. There is also a **C** command which continues execution but passes the signal that stopped the program back to the program. Use the **C** command for testing user-written signal handlers.

To tell **sdb** where to start running the program, specify a line number with the **g** command. For example:

```
*17 g
```

starts the program at line 17 of the current procedure. Use this command to avoid running a section of code that does not work. Do not start running in a different procedure than the one containing the breakpoint.

The **s** command runs a single line of source code. Use it to slowly run the program to examine its behavior in detail. The **S** command is like the **s** command but does not stop within called procedures. Use the **S** command to test the calling procedure when the called procedure works correctly.

Use the **m** command to monitor a specified memory location. The **sdb** program runs the program one line at a time until:

- The contents of the memory location change
- A specified number of steps are executed.

The **m** command is:

```
[variable]$m [count]
[address]:m [count]
```

The following examples show some uses of the **m**command:

```
*i$m
*i$m 10
*0x100bc:m
*0x100bc:m 5
```

If the count is omitted, **sdb** uses a value of infinity. The current procedure must be able to get to the variable. The **m** command is implemented in software and can be very slow.

To control how much information is displayed when single-stepping, set the level with the **v** command. The following examples show the range of information available:

```
*v        # show current file and procedure name
            this is the standard value used
*1 v      # also display each source line before execution
*2 v      # also display each assembler statement
```

Calling Procedures

Call any of the procedures of the program from **sdb** for testing individual procedures with different arguments, or for calling diagnostic procedures that format data to aid in debugging. There are two ways to call a procedure:

```
*proc(arg1, arg2,  . . . )
*proc(arg1, arg2,  . . . )/
```

The first way runs the procedure. The second way runs the procedure and writes the value that it returns. Use the second way for calling procedures. The value is written in decimal unless some other format is specified. Arguments to procedures may be integer, character or string constants, or values of variables that are accessible from the current procedure.

Note: If a procedure is called when the program is *not* stopped at a breakpoint (such as when a core image is being debugged) all variables are initialized before the procedure is started. This makes it impossible to use a procedure which relies on data from a core dump.

Debugging Assembler Language

The **sdb** program can also examine programs at the assembler language level. It can display the assembler language statements associated with a line in the source and place breakpoints at arbitrary addresses. The **sdb** program can also display or modify the contents of the machine registers.

Displaying Assembler Language Statements

To display the assembler language statements associated with line 25 in procedure `main`, use the command:

```
*main:25?
```

The **?** command is identical to the / command except that it displays from the memory space for the program code (text space) instead of the data space. The standard format for displaying text space is the **i** format, which interprets the assembler language instruction.

Absolute addresses may be specified instead of line numbers by adding a : (colon) to them. For example:

```
*0x1024:?
```

displays the contents of address 0x1024 in text space. Note that the command:

```
*0x1024?
```

displays the instruction corresponding to line 0x1024 in the current procedure. You can also set or remove a breakpoint by specifying its absolute address. The following example sets a breakpoint at address 0x1024:

```
*0x1024:b
```

To single-step by assembler instruction rather than source line, use the **i** command. The **i** command ignores the signal that stopped the program. There is also the **I** command, which causes the program to execute one assembler instruction at a time, but also passes the signal that stopped the program back to the program.

Manipulating Registers

The **x** command writes the values of all the registers and the process status word (**psw**).

To specify individual registers, append a % (percent sign) to their name. The next example changes the value of register r3 to 22:

```
*r3%!22
```

Use the following names when referring to the registers and pointers:

- Floating-point registers: f0, f2, f4, and f6
- Frame pointer: r4 or fp
- Argument pointer: r5 or ap

The **X** command displays the value of the Instruction Address Register (IAR) and the assembler language instruction at that location.

An Example of a Debug Session

Figure 8-1 shows the use of some **sdb** commands.

```
$ cat testdiv2.c
main(argc, argv, envp)
char **argv, **envp;
{
   int i;
   i = div2( -1 );
   printf( "-1/2 = %d\n" , i );
}
div2(i)
{
   int j;
   j = i>>1;
   return(j);
}
$ cc-g testdiv2.c
$ a.out
-1/2 = -1
$ sdb
No core image #Warning message from sdb
*/ div2        #Search for procedure "div2"
8:div2(i){    #It starts on line 8
*z            #Print the next few lines
8:div2(i)
9:{
10:    int j;
11:    j = i>>1;
12:     return(j);
13:}
*div2:b #Place a breakpoint at the beginning of "div2"
div2:11b      #sdb echoes proc name and line number
```

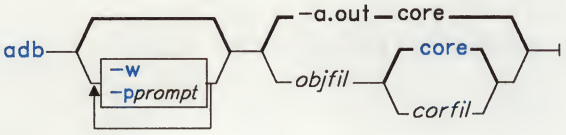
Figure 8-1 (Part 1 of 2). Example of Debug Session

```
*r              #Run the procedure
a.out           #sdb echoes command line executed
Breakpoint at #Executions stops just before line 11
div2:11:j = i>>1;
*t              #Print trace of subroutine calls
div2(i=-1)[testdiv2.c:11]
main(argc=1,argv=0x7fffff50,envp=0x7fffff58)[testdiv2.c:4]
*i/             #Print i
-1
*s              #Single step
div2:11:return(j)#Execution stops just before line 11
*j/             #Print j
-1
*10d             #Delete the breakpoint
*div2(1)/       #Try running "div2" with different arguments
0
*div2(-2)/
-1
*div2(-3)/
-2
*q
$
```

Figure 8-1 (Part 2 of 2). Example of Debug Session

The adb Program

The **adb** program provides a powerful set of commands to let you examine, debug, and repair executable binary files as well as examine non-ASCII data files. To use these commands, you must invoke **adb** from a shell command line and specify the file or files you want to debug. The command has the following syntax:



A5AC6002

Refer to *AIX Operating System Commands Reference* for detailed reference information about **adb**. The following sections explain how to start **adb** and describe the types of files available for debugging.

adb Command Summary

adb control commands:

$q	Exits **adb**.
$Q	Exits **adb**.
Ctrl D	Exits **adb**.

Program control commands:

address **[**, *count* **]** **:r** **[***arguments***]**
Execute program beginning at *address*; continue for *count* breakpoints; pass *arguments* to program.

address **[**, *count* **]** **:R** **[***arguments***]**
Execute program beginning at *address*; continue for *count* breakpoints; pass *arguments* to shell for expansion and then to program.

address **[**, *count* **]** **:b** **[***command***]**
Set breakpoint at *address* to stop program after it has been passed *count* times. Execute *command* when program stops.

address **:d**
Delete breakpoint at *address*.

address **[**, *count* **]** **:c** **[***signal***]**
Continue execution of program at *address* and ignore the next *count* breakpoints. Send *signal* to program when started.

address **[,** *count* **] :s**
Execute a single instruction at *address* and continue for *count* instructions.

Interrupt key
Stop program and return control to **adb**.

Quit key
Stop program and return control to **adb**.

:k
Kill program and return control to **adb**.

Display commands:

address **[,** *count* **]** = *format*
Display address.

address **[,** *count* **] ?** *format*
Display instructions.

address **[,** *count* **] /** *format*
Display value of variables.

.=
Display current address.

"=
Display last address that was used.

<*register*
Display value of *register*.

$b
Display breakpoints.

[, *count* **] $c**
Display *count* number of functions that have been called and have not yet returned control (active functions).

$r
Display the contents of CPU registers.

$e
Display external variables.

$m [*segment* **]**
Display the memory map for the specified *segment*.

$v
Display all nonzero variables in octal.

$p*string*
Use *string* as new prompt.

*n***$w**
Set output width to *n* characters.

Program editing commands:

*n***$s**
Set maximum allowed offset to *n*.

address **$d**
Set default input format to *address*.
$o
Set default input format to octal.
! *command*
Execute AIX Operating System *command*.
[*address* **] ?l** *value*
Find 2-byte *value* starting at *address*.
[*address* **] ?L** *value*
Find 4-byte *value* starting at *address*.
[*address* **] ?w** *value*
Write 2-byte *value* starting at *address*.
[*address* **] ?W** *value*
Write 4-byte *value* starting at *address*.
?m *segment-number file-position size*
Change text segment.
/m *segment-number file-position size*
Change data segment.

Starting with a Program File

You can debug any executable C or assembly language program file by typing a command line of the form:

```
adb [ filename ]
```

where *filename* is the name of the program file to be debugged. The **adb** program opens the file and prepares its text (instructions) and data for subsequent debugging. For example, the command:

```
adb sample
```

prepares the program named `sample` for examination and execution.

Once started, **adb** places the cursor on a new line and waits for you to type commands. It does not provide a prompt character unless you specify one with the **-p** option on the command line or with the **$p** command once in **adb**. If you start **adb** with the name of a file that does not exist or is in the wrong format, **adb** displays an error message first and waits for commands. For example, if you invoke **adb** with the command:

```
adb sample
```

and the file `sample` does not exist, **adb** displays the message:

```
sample: no such file or directory
```

–

You can also start **adb** without a file name. In this case, **adb** searches for the default file **a.out** in your current working directory and prepares it for debugging. Thus, the command:

```
adb
```

is the same as typing:

```
adb a.out
```

The **adb** program starts with the file **a.out** and waits for a command. If the **a.out** file does not exist, **adb** starts without a file and does not display an error message.

Starting with a Core Image File

The **adb** program also lets you examine the core image files of programs that caused fatal system errors. Core image files contain the contents of the CPU registers, stack, and memory areas of the program at the time of the error. Therefore, core image files provide a way to determine the cause of an error.

To examine a core image file with its corresponding program, you must give the name of both the core and the program file. The command line has the form:

```
adb programfile corefile
```

where *programfile* is the file name of the program that caused the error, and *corefile* is the file name of the core image file generated by the system. The **adb** program then uses information from both files to provide responses to your commands.

If you do not give a core image file, **adb** searches for the default core file, named **core**, in your current working directory. If such a file is found, **adb** uses it whether or not the file belongs to the given program. To prevent **adb** from opening this file, use the **mv** command to change the name of the core file. For example, the command:

```
mv core core.old
```

prevents **adb** from using the old core file for the current debugging session.

Starting with a Data File

You can use **adb** to examine data files by giving the name of the data file in place of the program or core file. For example, to examine a data file named `outdata`, type:

```
adb outdata
```

The **adb** program opens `outdata` and lets you examine its contents. This method of examining files is useful if the file contains non-ASCII data. The **adb** program provides a

way to look at the contents of the file in a variety of formats and structures. The **adb** command may display a warning when you give the name of non-ASCII data file in place of a program file. This usually happens when the content of the data file is similar to a program file. Like core files, data files cannot be executed.

Starting with the Write Option

If you open a program or data file with the **-w** option of the **adb** command, you can make changes and corrections to the file. For example, the command:

```
adb -w sample
```

opens the program file `sample` for writing. You may then use **adb** commands to examine and modify this file. The **-w** option causes **adb** to create a given file if it does not already exist. The option also lets you write directly to memory after executing the given program. For more information on the **-w** option, see "Patching Binary Files" on page 8-113.

Using a Prompt

When you start **adb** without specifying a prompt for **adb** to use, it starts with no prompt. The cursor simply moves to the next line on the display following the command and **adb** waits for your commands. Defining a prompt helps to indicate that **adb** is actually active. Use the **-p** option to define the prompt that **adb** uses. When used with the **-p** option, the **adb** command has the following form:

```
adb -pprompt filename
```

In this format *prompt* is any combination of characters. If the prompt includes spaces or special characters that are meaningful to the shell or to **adb** (such as >), enclose the prompt in quotation marks. For example, the command:

```
adb -p"ADB: " sample
```

sets the prompt to `ADB:` . Make sure there are no spaces between the **-p** and the new prompt; otherwise, **adb** tries to use the prompt as a file to debug, and displays an error message. The **adb** command does not supply a space at the end of the new prompt, so include a space at the end of the prompt string for clarity.

Leaving adb

You can stop **adb** and return to the system shell by using the **$q** or **$Q** commands. You can also stop **adb** by typing **Ctrl-D**. You cannot stop **adb** by pressing the Interrupt or Quit keys. These keys cause **adb** to wait for a new command.

Displaying Instructions and Data

The **adb** program provides several commands for displaying the instructions and data of a given program and the data of a given data file. The commands and their formats are:

Display address:

address [, *count*] = *format*

Display instruction:

address [, *count*] ? *format*

Display value of variable:

address [, *count*] / *format*

In this format, the symbols have the following meaning:

Symbol	Meaning
address	A value or expression that gives the location of the instruction or data item
count	An expression that gives the number of items to be displayed
format	An expression that defines how to display the items
=	Displays the address of an item
?	Displays the instructions in a text segment
/	Displays the value of variables

The following sections provide information to help you form addresses and expressions, choose formats, and display information when using **adb**. Where appropriate, the sections illustrate the information using a simple program, **adbsamp** as shown in Figure 8-2 on page 8-70.

```
char   str1[ ] = "This is a character string";
int    one     = 1 ;
int    number  = 456 ;
long   lnum    = 1234 ;
float  fpt     = 1.25 ;
char   str2[ ] = "This is the second character string";

main()
{
   one = 2;
}
```

Figure 8-2. Program Listing for adbsamp.c

If you are using the Programming Examples, this program is stored in the file **adbsamp.c**. Compile the program using the **cc** command to a file **adbsamp** with the following command:

```
cc adbsamp.c -o adbsamp
```

To start the debug session, type:

```
adb adbsamp
```

Forming Addresses

In **adb** addresses are 32-bit values that indicate a specific memory address. They can, however, be represented in the following forms:

Absolute address
The 32-bit value is represented by an 8-digit hexadecimal number, or its equivalent, in one of the other number base systems.

Symbol name
The location of a symbol defined in the program can be represented by the name of that symbol in the program. See "Using Symbols in Expressions" on page 8-77 for more information about symbol names.

Entry points

The entry point to a routine is represented by the name of the routine preceded by a period (.). For example, to refer to the address of the start of the routine called `main`, use the following notation:

```
.main
```

Displacements

You can refer to other points in program by using displacements from entry points in the program. For example, the following notation references the instruction that is four bytes past the entry point for `main`:

```
.main+4
```

Choosing Data Formats

A format is a letter or character that defines how data is to be displayed. The following are the most commonly used formats:

Letter	Format
o	1 word in octal (`int` variables)
O	2 words in octal (`long` variables)
d	1 word in decimal (`int` variables)
D	2 words in decimal (`long` variables)
x	1 word in hexadecimal (`int` variables)
X	2 words in hexadecimal (`long` variables)
u	1 word as an unsigned integer (`int` variables)
c	1 byte as a character (`char` variables)
s	A null-terminated character string (null-terminated arrays of `char` variables)
i	Machine instructions in mnemonic format
b	1 byte in octal (displays data associated with instructions or the high or low byte of a register)
a	The current symbolic address
n	A new line
r	A blank space
t	A horizontal tab.

For example, the following commands produce the indicated output when using the example program **adbsamp**:

adb Command	adb Response
`main=o`	`1560`
`main=O`	`4000001560`
`main=d`	`880`
`main=D`	`536871792`
`main=x`	`370`
`main=X`	`20000370`
`main=u`	`880`

A format can be used by itself or combined with other formats to present a combination of data in different forms. The following formats are usually combined with other formats to make the display more readable:

- `a`
- `n`
- `r`
- `t`

Data Format Examples

For example, the format:

```
main , 5 ? ia
```

causes the current address to be displayed after each instruction, producing the following output:

```
_main:          stc     r0,0
20000372:       bge     200002da
20000374:       sth     r0,0
20000376:       ble     _errno+22
20000378:       stc     r0,0
2000037a:
```

To make it clearer that the current address does not belong to the instruction that appears on the same line, add the newline format character to the command:

```
main , 5 ? ian
```

This instruction produces the following output:

```
_main:          stc     r0,0
20000372:
                bge     200002da
20000374:
                sth     r0,0
20000376:
                ble     _errno+22
20000378:
                stc     r0,0
2000037a:
```

In addition, you can put a number before a formatting character to indicate the number of times to repeat that format. For example, to print a listing of instructions and include addresses after every fourth instruction, use the following command:

```
main,3?4ian
```

This instruction produces the following output when used with the example program **adbsamp**:

```
_main:          stc     r0,0
                bge     200002da
                sth     r0,0
                ble     _errno+22
20000378:
                stc     r0,0
                bge     _str1+a
                lc      r6,3(r1)
                cas     r14,r6,r14
20000380:
                cas     r15,r7,r4
                sth     r7,0(r3)
                l       r6,10(r1)
                l       r2,10
20000388:
```

Be careful where you put the number. The following command, though similar to the previous command, does not produce the same output:

```
main,3?i4an
```

Instead, it produces the following output:

```
_main:          stc     r0,0
20000372:       20000372:       20000372:       20000372:
                bge     200002da
20000374:       20000374:       20000374:       20000374:
                sth     r0,0
20000376:       20000376:       20000376:       20000376:
```

You can combine format requests to provide elaborate displays. For example, the following command displays instruction mnemonics followed by their hexadecimal equivalent:

```
main,-1?i^xn
```

In this example, the display starts at the address `main`. The negative count (-1) causes an indefinite execution of the command, so that the display continues until an error condition (such as end-of-file) occurs. In the format, `i` displays the mnemonic instruction at that location, the `^` moves the current address back to the beginning of the instruction, and `x` re-displays the instruction as a hexadecimal number. Finally, `n` sends a newline character to the terminal. The output is similar to the following, only longer:

```
_main:          stc     r0,0
                1000
                bge     200002da
                1b4
                sth     r0,0
                2000
                ble     _errno+22
                33c
                stc     r0,0
                1000
                bge     _str1+a
                1d8
                lc      r6,3(r1)
                4361
                cas     r14,r6,r14
                6e6e
```

Displaying an Address (=)

Use the = (equal sign) to display an address in a given format. This command displays instruction and data addresses in simpler form and can display the results of arithmetic expressions. For example, the command:

```
main=an
```

displays the address of the symbol `main`:

```
20000370:
```

The following example shows a command that displays (in decimal) the sum of the internal variable `b` and the hexadecimal value `0x2000` together with its output:

```
<b+0x2000=D
                268443648
```

If a count is given, the same value is repeated that number of times. The following example shows a command that displays the value of `main` twice and the output that it produces:

```
main,2=x
                370     370
```

If no address is given, the current address is used. After executing the above command once (setting the current address to `main`), the following command repeats that function:

```
,2=x
                370     370
```

If you do not specify a format, **adb** uses the last format that was used with this command. For example in the following sequence of commands, both `main` and `one` are displayed in hexadecimal:

```
main=x
        370
one=
        33c
```

Using Integers in Expressions

When creating an expression, you can use integers in three forms: decimal, octal and hexadecimal. Decimal integers must begin with a nonzero decimal digit. Octal numbers must begin with a zero and have octal digits only. Hexadecimal numbers must begin with the prefix `0x` and may contain decimal digits and the letters `a` through `f` (in both uppercase and lowercase). The following are valid numbers:

Decimal	Octal	Hexadecimal
34	042	0x22
4090	07772	0xffa

Using Symbols in Expressions

Symbols are the names of global variables and functions defined within the program being debugged. Symbols are equal to the address of the given variable or function. They are stored in the program's symbol table and are available if the symbol table has not been stripped from the program file.

In expressions, you can spell the symbol exactly as it is in the source program or as it has been stored in the symbol table. Symbols in the symbol table are no more than eight characters long, and those defined in C programs are given a leading underscore (_). The following symbol pairs, taken from the sample program **adbsamp**, are valid symbols for the same address in the program:

Source Label	Symbol Table Label
main	_main
str1	_str1
lnum	_lnum

For example, the following commands to display the address of a symbol produce the same address:

```
main=d
                880
_main=d
                880
```

If the spelling of any two symbols is the same (except for a leading underscore), **adb** ignores one of the symbols and allows references only to the other. Therefore, do not use symbols in a source program that are similar in that manner. For example, if both `main`

and _main are in a source program, then **adb** accesses only the first one that appears in the source file and ignores the other.

When you use the **?** command, **adb** uses the symbols found in the symbol table of the program file to create symbolic addresses. Thus, the command sometimes gives a function name when displaying data. This does not happen if the **?** command is used for text (instructions) and the / command for data.

Local variables cannot be addressed. For example, the following attempt to display the local variable b while debugging the sample program **adbsamp2** results in the indicated error message from **adb**:

```
b=d
symbol not found
```

Using adb Variables

The **adb** program automatically creates a set of its own variables when it starts. These variables are set to the addresses and sizes of various parts of the program file as defined in the following table:

Variable	Content
b	Base address of the data segment
d	Size of data segment
e	Entry address of the program
m	Execution type (magic number)
s	Size of stack segment
t	Size of text segment.

The **adb** program reads the program file to find the values for these variables. If the file does not seem to be a program file, then **adb** leaves the values undefined.

To display the values that **adb** assigns to these variables, use the **$v** command. The command lists the variable names followed by their values in the current format. The command displays any variable whose value is not zero. If a variable also has a nonzero segment value, the variable's value is displayed as an address; otherwise, it is displayed as a number. The following example shows the use of this command to display the variable values for the sample program **adbsamp**.

```
$v
variables
b = 10000000
d = 130
e = 10000038
m = 108
t = 298
```

You can use the current value of an **adb** variable in an expression by preceding the variable name with a less than sign (<). The following example displays the current value of the base variable b:

```
<b=X
                10000000
```

You can create your own variables or change the value of an existing variable by assigning a value to a variable name with the greater than sign (>). The assignment has the form:

expression > variablename

where *expression* is the value to be assigned to the variable, and *variablename* is the variable to receive the value. The *variablename* must be a single letter. For example, the assignment:

```
0x2000>b
```

assigns the hexadecimal value 0x2000 to the variable b. Display b again to show that the assignment occurred:

```
<b=X
                2000
```

Finding the Current Address

The **adb** program has two special variables that keep track of the last address used in a command and the last address typed with a command. The dot (.) variable, also called the current address, contains the last address that was used in a command. The double quotation mark (") variable contains the last address that was typed with a command. The (.) and (") variables usually contain the same address except when implied commands, such as the newline and caret (^) characters, are used. These characters automatically increase and decrease the (.) variable but leave the (") variable unchanged.

Both the (.) and the (") variables can be used in any expression. The less than sign (<) is not required. For example, the following commands display these variables at the start of debugging with the **adbsamp** program:

```
.=
                0
"=
                0
```

Using Registers

To display the content of the processor registers, use the **$r** (or **$R**) command. For example after running the program **adbsamp** under **adb** control, the **$r** command produces the following listing:

```
mq     20003a24
cs     0
ics    4
pc     10000038  start
r15    10004f78
r14    0
r13    3fffe6a8
r12    3fffe3d0
r11    3fffe44c
r10    0
r9     20004bcc
r8     200041d8
r7     0
r6     200030bc
r5     20004c68
r4     3fffe6b0
r3     0
r2     0
r1     20000298
r0     20003224
start:          b       start+8
```

The registers displayed are defined in Figure 8-3 on page 8-82. The **adb** program lets you use the current value of the CPU registers in expressions. You can give the value of the register by preceding its name with the less than sign (<). The following example displays the content of the Instruction Address Register (pc):

```
<pc=X
                10000038
```

Register names cannot be used unless **adb** has been started with a core file, or the program is currently being run under **adb** control.

Name	Purpose
r0 - r15	General Purpose Registers, usually defined as: **r0** *pcp* - Constant pool pointer for called routine. **r1** *fp* - Caller's stack frame pointer **r2** *P1* - First word of passed parameters (if any) **r3** *P1* - Second word of passed parameters (if any) **r4** *P1* - Third word of passed parameters (if any) **r5** *P1* - Fourth word of passed parameters (if any) **r6 - r14** Data local to caller **r15** *link* - Return address
mq	*Multiplier Quotient* - Used by Multiply Step (**m**), Divide Step (**d**), Move to SCR (**mts**) and Move from SCR (**mfs**) instructions.
cs	*Condition Status* - Bits 24 through 31 hold status bits: **24** Permanent Zero (PZ) **25** Less Than (LT) **26** Equal (EQ) **27** Greater Than (GT) **28** Carry Zero (C0) **29** Reserved **30** Overflow (OV) **31** Test Bit (TB)
ics	*Interrupt Control Status* - Only bits 16 through 31 are used. The use is defined by the hardware processor being used. See *Hardware Technical Reference* for information about this register.
pc	*Program Counter* or *Instruction Address Register* - Contains the address of the next instruction to be executed.

Figure 8-3. Registers Displayed by adb

Using Operators in Expressions

You can combine integers, symbols, variables, and register names with the following operators:

Unary:

~	Not
-	Negative
*	Contents of location

Binary:

+	Addition
-	Subtraction
*	Multiplication
%	Integer division
&	Bitwise AND
\|	Bitwise inclusive OR
^	Modulo
#	Round up to the next multiple

adb uses 32-bit arithmetic. Values that exceed 2,147,483,647 (decimal) are displayed as negative values. The following example shows the results of assigning two different values to an **adb** variable n, and then displaying the value in both decimal and hexadecimal:

```
2147483647>n
<n=D
                2147483647
<n=X
                7fffffff
2147483648>n
<n=D
                -2147483648
<n=X
                80000000
```

Unary operators have higher precedence than binary operators. All binary operators have the same precedence and are evaluated in order from left to right. Thus, **adb** evaluates the following binary expressions as shown:

```
2*3+4=d
                10
4+2*3=d
                18
```

You can change the precedence of the operations in an expression by using parentheses. The following example shows how the previous expression is changed by using parentheses:

```
4+(2*3)=d
                    10
```

The unary * operator treats the given address as a pointer into the data segment. An expression using this operator changes to the value pointed to by that pointer. For example, the expression:

```
*0x1234
```

is equal to the value at the data address 0x1234, whereas:

```
0x1234
```

is just equal to 0x1234.

An Example of Simple Formatting

The following example shows how to combine formats in the **?** or **/** commands to display different types of values when stored together in the same program. It uses the program **adbsamp**. For the commands to have variables with which to work, you must first set a breakpoint to stop the program, and then run the program until it finds the breakpoint. Use the **:b** command to set a breakpoint:

```
.main+4:b
```

Use the **$b** command to show that the breakpoint is set:

```
$b
breakpoints
count   bkpt            command
1       .main+4
```

Run the program until it finds the breakpoint:

```
:r
adbsamp: running
breakpoint      .main+4:        li      r5,2
```

You can now display conditions of the program when it stopped. To display the value of each individual variable, give its name and corresponding format in a (/) command. For example, the following command displays the contents of `str1` as a string:

```
str1/s
_str1:
_str1:          This is a character string
```

The following command displays the contents of `number` as a decimal integer:

```
number/D
_number:
_number:        456
```

You can choose to view a variable in a variety of formats. For example, you can display the long variable **lnum** as a 4-byte decimal, octal, and hexadecimal number by using the commands:

```
lnum/D
_lnum:
_lnum:          1234

lnum/O
_lnum:
_lnum:          2322

lnum/X
_lnum:
_lnum:          4d2
```

You can also examine variables in other formats. For example, the following command displays some variables as eight hexadecimal values on a line and continues for five lines.

```
str1,5/8x
_str1:
_str1:      5468    6973    2069    7320    6120    6368    6172    6163
            7465    7220    7374    7269    6e67    0       0       1

_number:    0       1c8     0       4d2     3fa0    0       5468    6973
            2069    7320    7468    6520    7365    636f    6e64    2063
            6861    7261    6374    6572    2073    7472    696e    6700
```

Since the data contains a combination of numeric and string values, display each value as both a number and a character to see where the actual strings are located. You can do this with one command:

```
str1,5/4x4^8Cn
_str1:
_str1:          5468    6973    2069    7320    This is
                6120    6368    6172    6163    a charac
                7465    7220    7374    7269    ter stri
                6e67    0       0       1       ng~@~@~@~@~@~A

_number:        0       1c8     0       4d2     ~@~@~A~~@~@~D~
```

In this case, the command displays four values in hexadecimal, then displays the same values as eight ASCII characters. The caret (^) is used four times just before displaying the characters to set the current address back to the starting address for that line.

To make the display easier to read, you can insert a tab between the values and characters and give an address for each line:

```
str1,5/4x4^8t8Cna
_str1:
_str1:          5468    6973    2069    7320            This is
_str1+8:        6120    6368    6172    6163            a charac
_str1+10:       7465    7220    7374    7269            ter stri
_str1+18:       6e67    0       0       1               ng~@~@~@~@~@~A
_number:
_number:        0       1c8     0       4d2             ~@~@~A~~@~@~D~
_fpt:
```

Debugging Program Execution with adb

The **adb** program provides commands to control the execution of programs being debugged. The following sections explain how to use these commands and how to display the contents of memory and registers. Where appropriate, the sections illustrate the information using a simple program, **adbsamp2** as shown in Figure 8-4.

```
int     fcnt,loop_count;

f(a,b)
int a,b;
{
        a = a+b;
        fcnt++ ;
        return(a);
}

main()
{
        loop_count = 0;
        while(loop_count <= 100)
        {
           loop_count = f(loop_count,1);
           printf("%s%d\n", "Loop count is: ", loop_count);
           printf("%s%d\n", "fcnt count is: ", fcnt);
        }
}
```

Figure 8-4. Program Listing for adbsamp2.c

If you are using the Programming Examples, this program is stored in the file **adbsamp2.c**. Compile the program using the **cc** command to a file **adbsamp** with the following command:

```
cc adbsamp2.c -o adbsamp2
```

To start the debug session, type:

```
adb adbsamp2
```

The C language does not generate statement labels for programs. Therefore, you cannot refer to individual C statements when using the debugger. To use execution commands effectively, you must be familiar with the instructions that the C compiler generates and how those instructions relate to individual C statements. One useful technique is to create an assembler language listing of your C program before using **adb**. Then, refer to the listing as you use the debugger. To create an assembler language listing, use the **-S** option of the **cc** command (see *C Language Guide and Reference*). For example, to create an assembler language listing of the example program, **adbsamp2.c**, use the following command:

```
cc -S adbsamp2.c -o adbsamp2
```

This command creates a file, **adbsamp2.s**, that contains the assembler language listing for the program, and compiles the program to the executable file, **adbsamp2**.

Executing a Program

You can execute a program by using the **:r** or **:R** commands. The commands have the form:

```
[ address ][,count ] :r [ arguments ]

[ address ][,count ] :R [ arguments ]
```

In this format *address* gives the address at which to start execution; *count* is the number of breakpoints to skip before one is taken; and *arguments* is the command line arguments, such as file names and options, to pass to the program.

If you do not supply an *address*, **adb** uses the start of the program. To execute the program from the beginning type:

```
:r
```

If you supply a *count*, **adb** ignores all breakpoints until the given number have been encountered. For example, to skip the first five named breakpoints, use the command:

```
,5:r
```

If you provide arguments, separate them by at least one space each. The arguments are passed to the program in the same way the system shell passes command line arguments to a program. You can use the shell redirection symbols.

The **:R** command passes the command arguments through the shell before starting program execution. You can use shell wildcard characters in the arguments to refer to multiple files or other input values. The shell expands arguments containing wildcard characters before passing them to the program. This feature is useful if the program expects multiple file names. For example, the following command passes the argument `[a-z]*.s` to the

shell where it is expanded to a list of the corresponding file names before being passed to the program.

```
:R [a-z]*.s
```

The **:r** and **:R** commands remove the contents of all registers and destroy the current stack before starting the program. This operation kills any previous copy of the program that may be running.

Setting Breakpoints

To set a breakpoint in a program, use the **:b** command. Breakpoints cause execution to stop when the program reaches the specified address. Control then returns to **adb**. The command has the form:

address [, *count*] :b [*command*]

In this format *address* must be a valid instruction address; *count* is a count of the number of times you want the breakpoint to be skipped before it causes the program to stop; and *command* is the **adb** command you want to execute each time that the instruction is executed (regardless of whether the breakpoint stops the program).

Set breakpoints to stop program execution at a specific place in the program, such as the beginning of a function, so that you can look at the contents of registers and memory. For example, when debugging the example program, **adbsamp2**, the following command sets a breakpoint at the start of the function named `f`:

```
.f :b
```

The breakpoint is taken just as control enters the function and before the function's stack frame is created.

A breakpoint with a count is used within a function that is called several times during execution of a program, or within the instructions that correspond to a `for` or `while` statement. Such a breakpoint allows the program to continue to execute until the given function or instructions have been executed the specified number of times. For example, the following command sets a breakpoint for the second time that the function `f` is called in the program **adbsamp2**:

```
.f,2 :b
```

The breakpoint does not stop the function until the second time the function is executed.

Displaying Breakpoints

Use the **$b** command to display the location and count of each currently defined breakpoint. This command displays a list of the breakpoints by address and any count or commands specified for the breakpoints. For example, the following example sets two breakpoints in the file **adbsamp2** and then uses the **$b** command to display those breakpoints:

```
.f+4:b
.f+8:b$v
$b
breakpoints
count   bkpt            command
1       .f+8            $v
1       .f+4
```

When the program runs, it stops at the first breakpoint that it finds (at `.f+4`). If you use the **:c** command to continue execution, it stops again at the next breakpoint and executes the command **$v**. The command and response sequence looks like the following example.

```
:r
adbsamp2: running
breakpoint      .f+4:           st      r3,32(r1)
:c
adbsamp2: running
variables
b = 268435456
d = 236
e = 268435512
m = 264
breakpoint      .f+8:           l       r15,32(r1)
```

Deleting Breakpoints

Use the **:d** command to delete a breakpoint from a program. This command has the form:

address `:d`

In this format *address* is the address of the breakpoint to delete. For example, the following command deletes one of the breakpoints shown in the example for the **$b** command previously.

```
.f+4:d
```

Continuing Execution

Use the **:c** command to continue the execution of a program after it has been stopped by a breakpoint. The command has the form:

`[` *address* `][,`*count*`]` `:c` `[`*signal*`]`

In this format *address* is the address of the instruction at which to continue execution; *count* is the number of breakpoints to ignore; and *signal* is the number of the signal to send to the program (see **signal** in *AIX Operating System Technical Reference*).

If you do not supply an *address*, the program starts at the next instruction after the breakpoint. If you supply a *count*, **adb** ignores the first *count* breakpoints.

Single-Stepping a Program

Use the **:s** command to execute a program one instruction at a time (***single-step***). This command executes an instruction and returns control to **adb**. The command has the form:

`[`*address* `] [,` *count* `] :s`

In this format *address* is the address of the instruction you want to execute, and *count* is the number of times you want to repeat the command. If you do not supply an *address*, **adb** uses the current address. If you supply a *count*, **adb** continues to execute each successive instruction until *count* instructions have been executed. Breakpoints are ignored while single-stepping. For example, the following command executes the first five instructions in the function `main`:

```
.main,5:s
```

Stopping a Program with Interrupt and Quit

Use either the Interrupt or the Quit key to stop execution of a program at any time. Pressing either of these keys stops the current program and returns control to **adb**. These keys are useful with programs that have infinite loops or other program errors.

When you press the Interrupt or Quit key to stop a program, **adb** automatically saves the signal. If you start the program again using the **:c** command, **adb** automatically passes the signal to the program. This feature is useful when testing a program that uses these signals as part of its processing. To continue execution of the program without sending signals, use the command:

```
:c 0
```

The command argument 0 prevents a signal from being sent to the program.

Killing a Program

To kill the program you are debugging, use the **:k** command. This command kills the process created for the program and returns control to **adb**. The command clears the current contents of the CPU registers and stack and begins the program again. The following example shows the use of the **:k** command to clear the current process from **adb**:

```
:k
560:  killed
```

Displaying the C Stack Backtrace

To trace the path of all active functions, use the **$c** command. This command lists the names of all functions that have been called and have not yet returned control. It also lists the address from which each function was called and the arguments passed to each function. For example, the following command sequence sets a breakpoint at function address `.f+2` in **adbsamp2**. The breakpoint executes the command **$c**. The program is started, runs to the breakpoint and then displays a backtrace of the C language functions called:

```
.f+2:b$c
:r
adbsamp2: running
.f(0,0) .main+26
.main(0,0,0) start+fa
breakpoint      .f+2:           tgte    r2,r2
```

By default, the **$c** command displays all calls. To display fewer calls, supply a count of the number of calls to display. For example, the following command displays only one of the active functions at the preceding breakpoint:

```
,1$c
.f(0,0) .main+26
```

Displaying External Variables

Use the **$e** command to display the values of all external variables in the program. External variables are the variables in your program that have global scope or have been defined outside of any function. They include variables defined in library routines used by your program.

The **$e** command is useful to get a list of the names for all available variables or a summary of their values. The command displays one name on each line with the variable's value (if any) on the same line. The following example shows setting a breakpoint in **adbsamp2** that executes the **$e** command, and the output that results when the program runs (be sure to delete any previous breakpoints that you may have set):

```
.f+2:b$e
:r
adbsamp2: running
_errno:     0
_environ:   3fffe6bc
__NLinit:   10000238
_main:      100001ea
_exit:      1000028c
_fcnt:      0
_loop_count:            1
_f:         100001b4
_NLgetfile: 10000280
_write:     100002e0
__NLinit__X:            10000238
_NLgetfile__X:          10000280
__cleanup:  100002bc
__exit:     100002c8
_exit__X:   1000028c
__cleanup__X:           100002bc
breakpoint      .f+2:           st      r2,1c(r1)
```

An Example: Tracing Multiple Functions

Note: The example program used in this section, **adbsamp3**, contains an infinite recursion of subfunction calls. If you run this program to completion, it causes a memory fault error and quits.

The following example shows how to execute a program under **adb** control. In particular, it shows how to set breakpoints, start the program, and examine registers and memory. Figure 8-5 on page 8-97 shows the program listing for the program. If you are using the Programming Examples, the source program is stored in a file named **adbsamp3.c**. Compile this program to an executable file named **adbsamp3** using the **cc** command:

```
cc adbsamp3.c -o adbsamp3
```

To start the session and open the program file, use the following command (no core file is used):

```
adb adbsamp3
```

First set breakpoints at the beginning of each function using the **:b** command:

```
.f:b
.g:b
.h:b
```

Display their locations with the **$b** command:

```
$b
breakpoints
count   bkpt            command
1       .h
1       .g
1       .f
```

Next display the first five instructions in the **f** function:

```
.f,5?ia
.f:
.f:             b       .f+34
.f+2:           st      r2,24(r1)
.f+4:           st      r3,28(r1)
.f+6:           l       r0,24(r1)
.f+8:           l       r15,28(r1)
.f+a:
```

```
int     fcnt,gcnt,hcnt;
h(x,y)
int x,y;
{
        int hi; register int hr;
        hi = x+1;
        hr = x-y+1;
        hcnt++ ;
        hj:
        f(hr,hi);
}
g(p,q)
int p,q;
{
        int gi; register int gr;
        gi = q-p;
        gr = q-p+1;
        gcnt++ ;
        gj:
        h(gr,gi);
}
f(a,b)
int a,b;
{
        int fi; register int fr;
        fi = a+2*b;
        fr = a+b;
        fcnt++ ;
        fj:
        g(fr,fi);
}
main() {
        f(1,1);
}
```

Figure 8-5. Tracing Multiple Functions Example Program

Display five instructions in function **g** without their addresses:

```
.g,5?i
.g:
.g:             b       .g+34
                st      r2,24(r1)
                st      r3,28(r1)
                l       r0,28(r1)
                l       r15,24(r1)
```

Start the program with the following command:

```
:r
adbsamp3: running
breakpoint      .f:             b       .f+34
```

The **adb** program begins to execute the sample program until it reaches the first breakpoint and stops. Since execution to this point caused no errors, remove the first breakpoint:

```
.f:d
```

Continue the program:

```
:c
adbsamp3: running
breakpoint      .g:             b       .g+34
```

The **adb** program starts the program at the next instruction and continues until the next breakpoint where it stops. Trace the path of execution by entering:

```
$c
.g(0,0) .f+2a
.f(1,1) .main+e
.main(0,0,0) start+fa
```

The command displays a report that shows the three active functions: `main`, `f` and `g`. Display the contents of the integer variable `fcnt` by entering the command:

```
fcnt/D
_fcnt:
_fcnt:          1
```

Next continue execution of the program and skip the first 10 breakpoints by entering:

```
,10:c
adbsamp3: running
breakpoint      .g:             b       .g+34
```

The **adb** program starts the program and displays the running message again. It does not stop the program until exactly 10 breakpoints have been encountered. To show that these breakpoints have been skipped, display the backtrace again:

```
$c
.g(0,0) .f+2a
.f(2,11) .h+28
.h(10,f) .g+2a
.g(11,20) .f+2a
.f(2,f) .h+28
.h(e,d) .g+2a
.g(f,1c) .f+2a
.f(2,d) .h+28
.h(c,b) .g+2a
.g(d,18) .f+2a
.f(2,b) .h+28
.h(a,9) .g+2a
.g(b,14) .f+2a
.f(2,9) .h+28
.h(8,7) .g+2a
.g(9,10) .f+2a
.f(2,7) .h+28
.h(6,5) .g+2a
.g(7,c) .f+2a
.f(2,5) .h+28
.h(4,3) .g+2a
.g(5,8) .f+2a
.f(2,3) .h+28
.h(2,1) .g+2a
.g(2,3) .f+2a
.f(1,1) .main+e
.main(0,0,0) start+fa
```

Using the adb Memory Maps

The **adb** program prepares a set of maps for the text and data segments in your program and uses these maps to access items that you request for display. The following sections describe how to view these maps and how they are used to access the text and data segments.

Displaying the Memory Maps

Use the **$m** command to display the contents of the memory maps. The command displays the maps for all segments in the program. It uses information taken from either the program and core files or directly from memory. The **$m** command displays information similar to the following:

```
$m
? map       'adbsamp3'
b1 = 10000000        e1 = 10000370        f1 = 0
b2 = 20000370        e2 = 20000474        f2 = 370
/ map       '-'
b1 = 0               e1 = 0               f1 = 0
b2 = 0               e2 = 0               f2 = 0
```

The display defines memory mapping parameters for the text (b1, e1 and f1) and data (b2, e2 and f2) segments for the two files being used by **adb**. In this example, it shows values for **adbsamp3** only. The second set of map values are for the core file being used. Since none was in use, it shows the file name as **-**.

Changing the Memory Map

You can change the values of a memory map by using the **?m** and **/m** commands. These commands assign specified values to the corresponding map entries. The commands have the form:

```
?m b1 e1 f1

/m b1 e1 f1
```

The following example shows the results of these commands on the memory map displayed with the **$m** command in the previous example:

```
?m 10000100 10000470 0
/m 100 100 100
$m
/ map       'adbsamp3'
b1 = 10000100        e1 = 10000470        f1 = 0
b2 = 20000370        e2 = 20000474        f2 = 370
/ map       '-'
b1 = 100             e1 = 100             f1 = 100
b2 = 0               e2 = 0               f2 = 0
```

To change the data segment values, add a * (asterisk) after the / or ?.

```
?*m 20000270 20000374 270
/*m 200 200 200
$m
? map       'adbsamp3'
b1 = 10000100        e1 = 10000470        f1 = 0
b2 = 20000270        e2 = 20000374        f2 = 270
/ map       '-'
b1 = 100             e1 = 100             f1 = 100
b2 = 200             e2 = 200             f2 = 200
```

Combining Commands on a Single Line

You can give more than one command on a line by separating the commands with a semicolon (;). The commands are performed one at a time, starting at the left. Changes to the current address and format carry over to the next command. If an error occurs, the remaining commands are ignored. For example, the following sequence displays both the **adb** variables and then the active subroutines at one point in the program **adbsamp2**:

```
$v;$c
variables
b = 10000000
d = ec
e = 10000038
m = 108
t = 2f8
.f(0,0) .main+26
.main(0,0,0) start+fa
```

Creating adb Scripts

You can direct **adb** to read commands from a text file instead of the keyboard by redirecting the standard input file when starting **adb**. To redirect the standard input, use the input redirection symbol < and supply a file name. For example, use the following command to read commands from the file `script`:

```
adb sample <script
```

The file must contain valid **adb** commands. Use **adb** script files when the same set of commands can be used for several different object files. Scripts can display the contents of core files after a program error. For example, Figure 8-6 shows a file containing commands that display information about a program error. When that file is used as input to **adb** using the following command, it produces the output shown in Figure 8-7 on page 8-104:

```
$ adb adbsamp2 <script
```

```
120$w
4095$s
.f:b
:r
=ln"======= adb Variables ======="
$v
=ln"======= Memory Map ======="
$m
=ln"======= C Stack Backtrace ======="
$C
=ln"======= C External Variables ======="
$e
=ln"======= Registers ======="
$r
0$s
=ln"======= Data Segment ======="
<b,10/8xna
```

Figure 8-6. Example adb Script File

```
adbsamp2: running
breakpoint      .f:             b       .f+24

                ======= adb Variables =======
variables
b = 10000000
d = ec
e = 10000038
m = 108
t = 2f8

                ======= Memory Map =======
? map       'adbsamp2'
b1 = 10000000        e1 = 100002f8        f1 = 0
b2 = 200002f8        e2 = 200003e4        f2 = 2f8
/ map       '-'
b1 = 0               e1 = 0               f1 = 0
b2 = 0               e2 = 0               f2 = 0

                ======= C Stack Backtrace =======
.f(0,0) .main+26
.main(0,0,0) start+fa

                ======= C External Variables =======
_errno:     0
_environ:   3fffe6bc
__NLinit:   10000238
_main:      100001ea
_exit:      1000028c
_fcnt:      0
```

Figure 8-7 (Part 1 of 3). Example adb Script File Output

```
_loop_count:            1
_f:         100001b4
_NLgetfile: 10000280
_write:     100002e0
__NLinit__X:            10000238
_NLgetfile__X:          10000280
__cleanup:  100002bc
__exit:     100002c8
_exit__X:   1000028c
__cleanup__X:           100002bc
                ======= Registers =======
mq    20003a24  _errno+3634
cs    100000     gt
ics   1000004
pc    100001b4  .f
r15   10000210  .main+26
r14   20000388  _main
r13   200003ec  _loop_count
r12   3fffe3d0
r11   3fffe44c
r10   0
r9    20004bcc
r8    200041d8  _errno+3de8
r7    0
r6    200030bc  _errno+2ccc
r5    1
r4    200003ec  _loop_count
r3    f4240
r2    1
r1    3fffe678
r0    20000380  _f
.f:             b       .f+24
```

Figure 8-7 (Part 2 of 3). Example adb Script File Output

```
                ======= Data Segment =======
10000000:       103     5313    3800    0       0       2f8     0       ec
10000010:       0       10      1000    38      0       0       0       1f0
10000020:       0       0       0       0       1000    0       2000    2f8
10000030:       0       0       0       0       4       6000    0       6000
10000040:       6e10    61d0    9430    a67     6730    6820    c82e    8
10000050:       8df0    94      cd0e    60      6520    a424    a432    c84e
10000060:       8       8df0    77      cd0e    64      6270    8df0    86
10000070:       cd0e    60      6520    a424    a432    6470    8df0    6a
10000080:       cd0e    64      c82e    19      8df0    78      cd0e    60
10000090:       6520    a424    a432    c84e    19      8df0    5b      cd0e
100000a0:       64      cd2e    5c      7022    d408    64      911     c82e
100000b0:       2e      8df0    63      cd0e    60      6520    a424    a432
100000c0:       c84e    2e      8df0    46      cd0e    64      15      6280
100000d0:       8df0    60      cd0e    68      c82e    3f      8df0    4e
100000e0:       cd0e    60      6520    a424    a432    c84e    3f      8df0
100000f0:       31      cd0e    64      c820    14      8df0    2b      cd0e
10000100:
```

Figure 8-7 (Part 3 of 3). Example adb Script File Output

Setting Output Width

Use the **$w** command to set the maximum width (in characters) of each line of output created by **adb**. The command has the form:

```
n$w
```

In this format *n* is an integer that specifies the width in characters of the display. You can give any width convenient for your terminal or display device. The default width when **adb** is first invoked is 80 characters.

This command can be used when redirecting output to a line printer or special terminal. For example, the following command sets the display width to 120 characters, a common maximum width for line printers:

```
120$w
```

Setting the Maximum Offset

The **adb** program normally displays memory and file addresses as the sum of a symbol and an offset. This format helps to associate the instructions and data on the display with a particular function or variable. When **adb** starts up, it sets the maximum offset to 255, so that **adb** assigns symbolic addresses only to instructions or data that occur less than 256 bytes from the start of the function or variable. Instructions or data beyond that point are given numeric addresses.

In many programs, the size of a function or variable is actually larger than 255 bytes. For this reason **adb** lets you change the maximum offset to accommodate larger programs. You can change the maximum offset by using the **$s** command. The command has the form:

```
n$s
```

In this format *n* is an integer that specifies the new offset. For example, the following command increases the maximum possible offset to 4095:

```
4095$s
```

All instructions and data that are less than 4096 bytes away are given symbolic addresses. You can disable all symbolic addressing by setting the maximum offset to zero. All addresses are given numeric values instead.

Setting Default Input Format

To set the default format for numbers used in commands, use the **$d** or **$o** (octal) commands. The default format tells **adb** how to interpret numbers that do not begin with 0 (octal) or 0x (hexadecimal) and how to display numbers when no specific format is given.

Use these commands to work with a combination of decimal, octal, and hexadecimal numbers. For example, to be able to use octal (base 8) addresses without preceding each address with the 0 identifier, use the following command:

```
$o
```

After you enter that command, **adb** displays all numbers in octal format except those specified in some other format.

When you first start **adb**, the default format is hexadecimal. If you change the default format, you can restore it as necessary using the **$d** command.

```
$d
```

To set the default format to decimal, use the following command:

```
0xa$d
```

Using AIX Operating System Commands

You can execute AIX Operating System commands without leaving **adb** by using the **adb** escape command (!). The escape command has the form:

```
! command
```

In this format *command* is the AIX Operating System command you want to execute. You must provide any required arguments with the command. The **adb** program passes this command to the system shell that executes it. When finished, the shell returns control to **adb**. For example, to display the date, enter the following command:

```
! date
```

The system displays the date and restores control to **adb**.

Computing Numbers and Displaying Text

You can perform arithmetic calculations while in **adb** by using the equal (=) command. This command directs **adb** to display the value of an expression in a specified format. The command converts numbers in one base to another, double-checks the arithmetic performed by a program, and displays complex addresses in easier form. For example, the following command displays the hexadecimal number 0x2a as the decimal number 42:

```
0x2a=d
        42
```

Similarly, the following command displays 0x2a as the ASCII character * (asterisk):

```
0x2a=c
        *
```

Expressions in a command can have any combination of symbols and operators. For example, the following command computes a value using the contents of the **r0** and **r1** registers and the **adb** variable b.

```
<r0-12*<r1+<b+5=X
                8fa86f95
```

You can also compute the value of external symbols to check the hexadecimal value of an external symbol address, as in the command:

```
main+5=X
                2000038d
```

The (=) command can also display literal strings. Use this feature in **adb** scripts to display comments about the script as it performs its commands. For example, the following command spaces three lines and then prints the message C Stack Backtrace on the terminal:

```
=3n"C Stack Backtrace"
```

An Example: Directory and Inode Dumps

This example shows how to create **adb** scripts to display the contents of a directory and the inode map of a AIX Operating System file system. In the example the directory is named **dir** and contains a variety of files. The AIX Operating System file system is associated with the device file /dev/hd3 (**/tmp**) which has the necessary permissions to be read by the user.

To display a directory, create an appropriate script. A directory normally contains one or more entries. Each entry consists of an unsigned inode number (inumber) and a 14-character file name. You can display this information by including a command in your script file. For example, the following command displays the first 20 entries separating the inode number and file name with a tab:

```
0,20?ut14cn
```

You can change the second number 20 to specify the number of entries in the directory. If you place the following command at the beginning of the script, **adb** displays the strings as headings for each column of numbers:

```
="inumber"8t"Name"
```

Once you have created the script file, redirect it as input when you start **adb** with the name of your directory. For example, the following command starts **adb** on directory **geo** using command input from the script file **ddump**:

```
adb geo - <ddump
```

The hyphen (-) prevents **adb** from opening a core file. The **adb** program reads the commands from the script file. The resulting display is shown in Figure 8-8 on page 8-111.

To display the inode table of a file system, create a new script and then start **adb** with the file name of the device associated with the file system. The inode table of a file system has a complex structure. Each entry contains:

- A word value for the file's status flags
- A byte value for the number links
- 2-byte values for the user and group IDs
- A byte and word value for the size
- 8-word values for the location on disk of the file's blocks
- 2-word values for the creation and modification dates.

```
                inumber Name
0:              26      .
                2       ..
                27      .estate
                28      adbsamp
                29      adbsamp.c
                30      calc.lex
                31      calc.yacc
                32      cbtest
                68      .profile
                66      .profile.bak
                46      adbsamp2.c
                52      adbsamp2
                35      adbsamp.s
                34      adbsamp2.s
                48      forktst1.c
                49      forktst2.c
                50      forktst3.c
                51      lpp_name
                33      adbsamp3.c
                241     sample
                198     adbsamp3
                55      msgqtst.c
                56      newsig.c
```

Figure 8-8. Example Directory Dump Output

The inode table starts at the address 02000. You can display the first entry by putting the following command in your script file:

```
02000,-1?on3bnbrdn8un2Y2na
```

The command specifies several new-line characters for the output display to make it easier to read.

To use the script file with the inode table of /dev/hd3, enter the following command:

```
adb /dev/hd3 - <script
```

Each entry in the display has the form:

```
02000: 073145
       0163  0164 0141
       0162  10356
       28770 8236 25956 27766 25455 8236 25956 25206
       1976 Feb 5 08:34:56  1975 Dec 28 10:55:15
```

Patching Binary Files

You can make corrections or changes to any file, including executable binary files, by invoking **adb** with the **-w** option and using the **w** and **W** commands. The following sections describe how to locate and change values in a file.

Locating Values in a File

You can locate specific values in a file by using the **l** and **L** commands. The commands have the form:

```
?l value
```
or
```
/l value
```

The search starts at the current address and looks for the expression indicated by *value*. The **l** command searches for 2-byte values. The **L** command searches for 4-byte values.

The **?l** command starts the search at the current address and continues until the first match or the end of the file. If the value is found, the current address is set to that value's address. For example, the following command searches for the first occurrence of the symbol `f` in the file **adbsamp2**:

```
?l .f
.write+a2
```

The value is found at `.write+a2`; the current address is set to that address.

Writing to a File

You can write to a file by using the **w** and **W** commands. The commands have the form:

```
[ address ] ?w value
```

In this format *address* is the address of the value you want to change, and *value* is the new value. The **w** command writes 2-byte values. The **W** command writes 4-byte values. For example, the following commands change the word `This` to `The`:

```
?l 'Th'

?W 'The'
```

The **W** changed all four characters.

Making Changes to Memory

You can also make changes to memory whenever a program has been executed. If you have used an **:r** command with a breakpoint to start program execution, subsequent **w** commands cause **adb** to write to the program in memory rather than to the file. This command is useful to make changes to a program's data as it runs, such as temporarily changing the value of program flags or constants.

Chapter 9. Installing and Updating a Program

CONTENTS

About This Chapter 9-2
Understanding System Guidelines 9-3
 Protecting System Directories 9-3
 Providing User Documentation 9-4
 Installing a Program in a Distributed Services Environment 9-5
Using Installation and Update Services 9-6
 Commands 9-6
 Internal Commands 9-7
What You Need to Install a Program 9-9
 Control Files 9-9
 Program Files 9-10
 Creating the Installation Procedure 9-11
What You Need to Update a Program 9-15
 Using the updatep Program 9-18
 Example Update Procedure 9-19
Installing and Updating Active Files 9-21
 Installing and Updating /bin/sh or /etc/restore 9-23
Allowing for Recovery 9-24
 Recovering from Kernel or Configuration Changes 9-24
 Sequencing Configuration File Updates 9-25
Creating the Program History File 9-26
Creating the Program Requirements File 9-29
 Using the Requirement String 9-30
 Requirements File Example Entry 9-31
Creating the Program Name File 9-33
Creating an Apply List File 9-34
Creating an Archive Control File 9-35
Creating a Special Requirement File 9-36
 Dependent Program Entry 9-36
 Changes to Report Templates Entry 9-37
 IPL Required Entry 9-37
 Changes to Configuration Entry 9-37
 Changes to Kernel Rebuild Entry 9-37
 Changes to VRM Entry 9-38
Creating a Save and Recover Directory 9-39
Designing Programs for a Code-Service Environment 9-43
 Code-Service Types 9-44

About This Chapter

After finishing the development and testing of a program, you must prepare it to be installed on other systems. This chapter describes the operating system services that help simplify installation procedures for people using the program. It describes:

- What the installation services can do
- What an installation program must do
- What files to include on the distribution diskette
- The format of each of the needed files.

This chapter also discusses things to consider in designing a program to run efficiently in a code-server environment.

At a later time you may need to change the program to provide additional features, or to correct problems encountered in actual usage of the program. This chapter also describes the operating system services that help change an installed program. It describes:

- What the update services can do
- What an update program must do
- What files to include on the update diskette.

Note: If your program includes support for a new device or added features for a currently supported device, you must also provide information to be added to the system configuration files. This information tells the system how to interface with the new driver support. Refer to *Device Driver Development Guide* for information about the type of configuration information you must supply.

Understanding System Guidelines

When preparing your program to install on the RT system, be aware of how the system files are organized so that the program can work as an integral part of the whole system. The following paragraphs describe some of the system conventions that any new program should follow.

Protecting System Directories

Each of the directories supplied with the system has a defined purpose. The permissions, owner and group ID for these directories are set so that the directories work properly with the system programs that use them.

CAUTION
Do not change the permission, owner or group id of any system file or directory. Changing these permissions prevents the programs supplied with the system from operating properly.

Many of the administrative system functions can be performed by a member of the system group without having to log in as root or know the root password. Refer to the description of each command in *AIX Operating System Commands Reference* to determine which functions the system group can perform. Some sensitive commands are still reserved for the person with superuser authority.

In addition, reserve the following directories for their intended purpose only:

/ Do not add files or mount file systems to subdirectories in the root file system without an urgent need to do so. The root file system contains fixed system directories and should not be used for application programs or user files. For additional constraints see "Server-Exclusive Programs" on page 9-44.

/usr Do not add files or directories to the **/usr** directory without an urgent need to do so. The **/usr** directory contains files and directories that the system uses. For additional constraints see "Server-Exclusive Programs" on page 9-44.

/usr/bin This directory contains common user executable files and is part of the default search path for all users. Put a short binary program or shell script in this directory that calls your large executable files. Put the large executable files in the directory **/usr/lpp/***pgm-name* (*pgm-name* is the identifier you choose for your program, as described in "What You Need to Install a Program" on page 9-9).

/usr/lpp	Create a directory in this directory to store your programs as described in "What You Need to Install a Program" on page 9-9. Do not put configuration information in **/usr/lpp/***pgmname*. For additional constraints see "Server-Exclusive Programs" on page 9-44.
/usr/lib	Use this directory to store those library files that have general use to all users. Store other library files in your directory, **/usr/lpp/***pgm-name*.
/usr/include	Use this directory to store those include files that have general use to all users. Store other include files in your directory, **/usr/lpp/***pgm-name*.

Providing User Documentation

You should provide documentation along with your program to enable the user to easily find any needed information, and to install and run your program properly. Pattern installation procedures after the format and steps shown in *Installing and Customizing the AIX Operating System* so that the user can use familiar procedures when installing any program on the system. Refer to information in *AIX Operating System Technical Reference* about creating device drivers and about the **vrcppr** routine when designing the installation. In addition to the normal installation and use information, you should provide information about your program's use of disk space in each of the following directories:

- **/usr**
- **/usr/bin**
- **/usr/lib**
- **/usr/lpp**
- **/usr/include**

Give usage information in terms of the number of 512-byte blocks that the program uses in each of the directories. The information for the kernel and operating system VRM is in *Installing and Customizing the AIX Operating System*. The usage information helps the person installing your program determine if there is enough space on the file system to load the program. If there is not enough space, refer the user to *Managing the AIX Operating System* for information about expanding an existing file system. *Installing and Customizing the AIX Operating System* contains information about expanding the VRM and page space minidisks.

Installing a Program in a Distributed Services Environment

Note: Installing a program in a Distributed Services environment can allow it to be executed from any of several RT work stations. Some programs may have licensing restrictions that prevent the program from being used in a distributed environment. Be careful not to violate the license agreement when installing a program in a Distributed Services environment.

When using Distributed Services and installing a program on one node (in a non code service environment) so that it can be executed from other nodes, be aware of the following points about how Distributed Services operates:

- The installation procedure for a program can do more than put a single command into a single directory. Frequently, it puts many files in different directories. Be sure the person installing your program is aware that the remote systems that use the program must issue the necessary **mount**s to put those files in the proper places in the remote systems' file tree. In some cases (to ensure that the files are present in case of a network failure, or for performance tuning) the administrator may keep copies of the files on each system that uses those files.
- The installation procedure performs prerequisite checking while installing the program, and verifies the presence of needed programs within the file tree that exists *at the time of installation.* Any remote system that uses the program must ensure that all required programs are in the correct places in *the remote system's* file tree.
- If the local system has mounted remote file systems when the program is installed, and those remote file systems provide needed prerequisite programs, then those remote file systems must always be mounted when the program is run.
- To speed up searches for the program, each of the remote systems that use the program should install an entry in their local **/bin** of **/usr/bin** directory that invokes the remote program.

For information about code service see "Designing Programs for a Code-Service Environment" on page 9-43.

Using Installation and Update Services

The installation and update services require only a few simple steps to install the program on the system. Typically, the person that installs the program only needs to put the first installation diskette in the diskette drive and enter the command:

```
installp
```

From that point on, messages to the screen tell the person when to change diskettes and what the status of the installation is.

In addition to providing an easy interface for the person installing the program or update, installation and update services also help ensure that the installation is correct. They:

- Maintain an accurate record of the revision state of the program on the system
- Check the revision level of other needed programs to ensure that they will work with the program
- Provide instructions to install the program
- Provide online copies of changes to documentation.

Commands

Installation and update services provides the following commands to be entered from the command line to install or update a program. Complete information about the syntax and usage of these commands is in *AIX Operating System Commands Reference.*

Command	Description
installp	Installs the program or programs on the diskette in the specified drive, according to an installation program on the installation diskette.
updatep	Installs one or more changes to an installed program. Changes must be in **backup** format, and can be in a file on disk, on tape, or on a diskette in the specified drive. Changes can be installed on a conditional basis to try them out. At a later time, you can either accept the changes for permanent installation or reject them. Rejecting the changes returns the changed program to its condition before the changes were installed.

Figure 9-1 (Part 1 of 2). Install and Update Commands

Command	Description
bffcreate	Creates a file in backup format (BF file) for use in a code-service environment. The output from **bffcreate** can be: • **update** — One file consisting of all the files from an **updatep** distribution media • **install:** – One file consisting of all the files from a complete program **installp** distribution media – One file for each program subset from an **installp** distribution media. • **non-standard** — One file consisting of all the files from a distribution media that does not follow **installp** conventions.
chngstate	This AIX command is used in a code-service environment to change the system state to either active service or stand-alone. The routine is used by the system initialization functions or by a system administrator. The routine validates and processes the code-server attribute file and installs any program upgrades needed for client and server compatibility in order to attach to a server as an active client. If a timeout error is encountered, **chngstate** goes to the next permitted attribute file entry and attempts to process it. Any other error causes the system to come up in stand-alone mode.
chkcomp	This AIX command is invoked from the **chngstate** process to check for client and server compatibility before allowing active service. Any incompatibilities between version, release, or level cause output records to be generated for use by **chngstate**, **installc**, or **updatec**.
setrdperm	This AIX command is used locally on a code server to manipulate the read permissions of files in server directories that are mounted over client directories.

Figure 9-1 (Part 2 of 2). Install and Update Commands

Internal Commands

Once either of the commands is running, the installation or update program can use the internal commands to access the installation and update services. Use these commands like shell commands. If the installation program is a shell procedure, use the commands like any other command. If the installation program is a C language program, use the **fork** and **exec** system calls, or the **system** subroutine to run the commands. Detailed information about the internal commands is in *AIX Operating System Commands Reference*. The internal commands do the functions outlined in Figure 9-2 on page 9-8.

Command	Description
/etc/ckprereq	Checks the revision level of programs.
/etc/codeserve/installc	Used by the **chngstate** process to install programs or program subsets on active service-clients.
/etc/codeserve/updatec	Used by the **installc** process to make an active-service client fully compatible with a specific server on an active-service network.
/etc/cvid	Backs up the VRM minidisk. Use this command to make a set of installation diskettes before making any changes to the VRM minidisk. You can then recover the previous version from these diskettes if the changes cause an error.
/etc/errupdate	Adds, replaces, or deletes the error report format templates in the file **/etc/errfmt**.
/etc/inudocm	Gets copies of update instructions or book changes to look at or print.
/etc/inurecv	Recovers all files and archived member files that a previous execution of **inusave** saved. This command also recovers any other saved files recorded in the configuration list file, or active list file.
/etc/inurest	Does simple restores and archives.
/etc/inusave	Saves some, or all, of the files and archived member files that will be replaced or modified when the program is installed or updated.
/etc/inuupdt	Applies a maintenance update for a single program.
/etc/mvmd	Does one of the following actions to a file on a VRM minidisk. • Adds a file • Deletes a file • Replaces an existing file • Changes file permissions • Moves the position of a file entry in a VRM directory listing.
/etc/trcupdate	Adds, replaces, or deletes trace report format templates in the file **/etc/trcfmt**.

Figure 9-2. Internal Commands

What You Need to Install a Program

To install a program on the RT system using the **installp** command, provide a set of control and program files in **backup** format on the installation media (either diskette or magnetic tape). The control files (see "Control Files") must appear before the program files (see "Program Files" on page 9-10).

Put the control and program files on the installation media using the **backup -i** command (backed up *by name*). See the description of this command in *AIX Operating System Commands Reference*. In addition, when backing up the files, use a relative path name with respect to the / directory on the installed system. For example, to create an installation diskette for a single program file (**ftrn**) to be stored in the **/bin** directory on the target system, provide an ordered list of files to the **backup -i** command:

```
./lpp_name
./usr/lpp/FORTRAN/liblpp.a
./bin/ftrn
```

The first two files are control files; the last file is the program file (or a group of files). This is the minimum required list of files.

Control Files

Provide the following control files at the beginning of the installation media:

Note: These files must have read-by-other permissions; the programs (installation scripts) must have execute-by-other permissions.

- A file named **lpp_name** that contains the title of the program or programs on the installation media (see "Creating the Program Name File" on page 9-33).
- A library file named **./usr/lpp/*pgm-name*/liblpp.a** (***pgm-name*** is a name for the program with a maximum of 8 lowercase, alphabetic characters). Create this library using the **ar** command. The library contains the following files:

 lpp.hist A file for logging changes to the program (see "Creating the Program History File" on page 9-26).

 instal The installation program that installs special features and files that the program uses. This program must be either a shell procedure or an executable program in **a.out** format.

 al An apply list file that contains the relative path names with respect to the / directory of all files to be restored.

 copyright An optional file that contains appropriate copyright information for this program.

prereq An optional file that lists the programs that the program uses and that, therefore, must be installed on the system (see "Creating the Program Requirements File" on page 9-29).

lpp.acf An optional file that defines archiving procedures for the program (see "Creating an Archive Control File" on page 9-35).

config An optional executable file that contains a procedure to update the system configuration. This is **not** the customization helper program specified in **/etc/master**. If the program does not do any customization, the **config** file is not needed. The procedure may be either a shell procedure, or a compiled procedure in **a.out** format.

lpp.doc An optional file containing changes to the book for the program.

Other files Other optional files can be added to the end of the archive file as needed by special installation programs that you decide to use.

- A library file named **./usr/lpp/***pgm-name***/liblpp.cp.a**. This is an archive file similar to **liblpp.a** that is only needed in a code-service network. The file contains program subset information and is only present if a program subset is present. For more information see "Client-Partial Programs" on page 9-46.

The **installp** program restores the library file **./usr/lpp/***pgm-name***/liblpp.a** and extracts the files in it. Then **installp** executes your **instal** program and passes it a single parameter that specifies the device containing the restore files. Access this parameter with `$1` if using a shell procedure or with `argv[1]` if using a C language program:

```
main(argc,argv)
int argc;
char *argv[];
```

When the **instal** program returns to **installp**, **installp** removes all files that do not begin with `lpp` from the **/usr/lpp/***pgm-name* directory. However, directories in that directory are not deleted.

Program Files

The program files are the files needed to run the program. Include these files after the needed control files in **backup -i** format.

Creating the Installation Procedure

In the archive file **./usr/lpp/***pgm-name***/liblpp.a**, provide an executable file, named **instal** to install the program and its files. The **installp** program executes this program during the installation procedure after it completes its preliminary tasks.

The installation program can then do some of the following tasks to complete the installation of the program.

1. Ensure that the needed level of other programs are installed on the system using **ckprereq**.
2. Perform active file processing if any of the files to be restored are being used. See "Installing and Updating Active Files" on page 9-21.

 CAUTION

 Do not use inusave when installing large programs that will fill the disk with the backup files.
3. Use **inusave** to back up any files that will be replaced.
4. Use **inurest** to restore files listed in the apply list file, and to archive constituent files.
5. Customize the system for the program. See *AIX Operating System Technical Reference* and *Device Driver Development Guide* for information to help you customize the system.
6. If an error occurs and **inusave** was previously used, then use **inurecv** to recover the previous state of the system.
7. If **inusave** was used, delete the directory, **/usr/lpp/***pgm-name***/inst_updt.save**.
8. Return a completion code to the **installp** program.

The installation procedures for complete programs and program subsets are usually the same.

Restoring the Program Diskette

After the installation program determines that the required programs are on the system, it can use the **inurest** command to restore the program diskette. For example, to restore the files listed in the apply list file **al** for the program named **myprog**, use a shell command in the following format:

```
/etc/inurest -qd device /usr/lpp/myprog/al myprog
```

This command restores all files from the specified device. The installation program uses the device passed to it by the **installp** program. The parameter `/usr/lpp/myprog/al` is the full path name of the apply list file, and `myprog` is the name of the program being installed.

Also use the **inurest** command to archive and then delete any restored archive member files. To enable the archive process, include a file **lpp.acf** in the library **./usr/lpp/***pgm-name***/liblpp.a** on the product diskette. See "Creating an Archive Control File" on page 9-35 for the format of this file. If this file is present, **inurest** archives and then deletes all restored files that are listed in both **lpp.acf** and the apply list file (**al** in the example).

Quiescing the System

When installing the program, the **installp** and **updatep** programs instruct the user to work with a system that has just been started and has no other users or user programs running. The **instal** or **update** procedure expects that the system is in a quiet state.

It is not always possible to have a quiet system in a code service environment. In this case, the **-q** (quiet) flag on **updatep** and **installp** suppresses most of the interaction queries.

Allowing for Individual Needs

The installation program may need to do some special tasks to ensure that the program operates properly. Special tasks to take care of include the following:

- Use the **inurest** command together with an apply list file and archive control file to store files in a library.
- Display progress messages during the different stages of the installation procedure.
- Recover from errors, or back out of the installation if errors occur.

Customizing the System for a Program

If a program requires changes to the system configuration, such as adding a new device driver, include a procedure to do this customization task. This procedure must be called **config** and must be in the archive file **./usr/lpp/***pgm-name***/liblpp.a** when the program is installed. Create an input file containing the stanzas to be added to the customization files.

The stanzas may include a device stanza for the device and device driver stanzas associated with the device for both the operating system VRM and the kernel. Refer to *Device Driver Development Guide* for more information about providing configuration information to the system. Steps you may need to include in your **config** procedure are:

- Archive any new device driver code into the kernel libraries.
- Back up system files that may be changed, such as:
 - **/etc/master**
 - **/etc/system**
 - **/etc/predefined**
- Restore any **/etc/ddi** files which may be associated with the new device.

- Use the **cfgcopsf** routine to open the stanza file.
- Use the **cfgcrdsz** routine to read each stanza from the stanza file.
- Use the **cfgddev** routine to delete device stanzas from **/etc/system**, if appropriate. If the program is reinstalled, remove previous device stanzas before adding the same device stanzas.
- Use the **cfgadev** routine to add device stanzas to **/etc/system** and to add operating system stanzas to **/etc/master**. The **cfgadev** routine receives a pointer to a device stanza to add to **/etc/system**, and pointers to an operating system device driver stanza. It adds each of these device driver stanzas to **/etc/master**.
- Use the **cfgcadsz** routine to add any device stanzas to **/etc/predefined**. If the new device associated with the program can be deleted from the system, adding the device stanza to **/etc/predefined** allows users to add the device again by using the **devices** command.
- Use the **cfgcclsf** routine to close the input file.
- Return a completion code to **instal**.

Refer to *AIX Operating System Technical Reference* for detailed information about the subroutines and file formats for customizing the system files.

Sending a Return Code to installp

When the installation program completes, it must send a return code to the **installp** program. This return code signals the end of the install program, and determines what actions the **installp** program will do next. Use one of the following values for a return code:

Code	Description
0	Successful completion. No additional action is needed.
2	Successful completion. The **installp** program updates superblocks, the i-node list, and flushes the buffers (**sync**). Then it performs an IPL on the operating system.
3	Successful completion. The **installp** program uses the **cfgaply** routine to build a new kernel. Then it updates superblocks, the i-node list, and flushes the buffers. Then it instructs the user to IPL the system and shuts down the operating system.

Figure 9-3 (Part 1 of 2). Return Codes to installp

Code	Description
4	Successful completion. The **installp** program uses the **cfgaply** routine to build a new kernel. Then it updates superblocks, the i-node list, and flushes the buffers before it performs an IPL on the operating system.
5	Installation cancelled by the **install** procedure without errors.
6	Successful completion. The **installp** program updates superblocks and the i-node list, and flushes the buffers (**sync**). Then it instructs the user to IPL the system and shuts down the operating system.
other	Error. The **installp** program sets the `Version`, `Release` and `Level` fields of the last information record in **lpp.hist** to all zeros, and writes the return code value in **lpp.hist** as an error code.

Figure 9-3 (Part 2 of 2). Return Codes to installp

What You Need to Update a Program

Note: When providing an update for a program that has had previous updates, ***all*** previous updates since the last release of the program must also be supplied on the update media to ensure that the current update can be installed successfully.

To update a program on the RT system using the **updatep** command, supply the following files on the first diskette of the product update. All files are in the format used by the **backup** command, and must be backed up by **name**, using a relative path name with respect to the / directory on the target system. The files must be in the following order:

1. An optional file named **./copyright** that contains a copy of the copyright information of each program on the diskette. If this file is present, it must be the first file on the diskette. It is referenced by the external diskette copyright label if there is not enough space on the label to include all of the appropriate copyright notices.
2. A file named **./lpp_name** that contains the name and title of each program (see "Creating the Program Name File" on page 9-33)
3. A library named **./usr/sys/inst_updt/control** and created with the **ar** command that contains the following files:

 *pgm-name*_**vrl**
 : A file for each program, or program subset, to be updated that contains the version, release and level numbers of the program after the update has been applied. The format of the one-record file is:

     ```
     VV RR LLLL
     ```

 which is similar to all version numbers used in this section, except that there are spaces between the version, release, and level numbers instead of periods. The record ends with a new-line character.

 service_num
 : An optional file that contains the Corrective Service Number. This number corresponds to the number which appears on update diskettes. If this file is present, **updatep** uses its content.

 lppsize
 : An optional file that contains an entry for each program that is being updated. Each entry contains:

 pgm-name usrsize pgmsize tmpsize

 The *usrsize* and *pgmsize* parameters specify the size of the update for the indicated program in 512-byte blocks. The two parameters must be separated by a blank. Each entry ends with a single new-line character.

 The **updatep** program checks the *usrsize* information to determine if there is enough space in the **/usr** file system to save the old versions of the programs

before beginning the update procedure. If the size of the added programs is larger than the available free space in the **/usr** file system, **updatep** gives the user the following options:

- Stop the update
- Continue with the update.

If the user continues, **updatep** does not save the current versions of the files and automatically commits the update. The previous version of the program cannot be recovered.

The **updatep** program uses *tmpsize* to determine if there is enough space in the **/tmp** file system to archive constituent files into the necessary library. The **ar** command uses **/tmp** when archiving files. It requires twice the number of existing blocks in a library to archive its files. If the calculated size of **/tmp** is too small, **updatep** will stop the update and issue an informational message.

Note: The **updatep** program also checks the space available on the **/(root)filesystem** for programs requiring a rebuild of the kernel. If the space available is too small, **updatep** will issue an informational message and stop the update.

Although this file is optional, you should include an **lppsize** file in all program updates. If **updatep** does not find this information, it does not check to see if there is enough space to save the previous version of the program.

*pgm-name*_**instr**
An optional library, one for each program being updated, that contains a set of files that contain instructions. If you do not provide instruction files, do not include this library. These files are standard text files. They are named according to the update level to which they apply, according to the format:

```
ui.VV.RR.LLLL
```

In this format the symbols VV, RR and LLLL represent the version, release, and level numbers. You can have instruction files for one or more of the program levels that are being updated.

*pgm-name*_**erata**
An optional library, one for each program being updated, that contains a set of files that contain changes to the book(s) for the program. If you do not provide changes to the books, do not include this library. These files are standard text files. They are named according to the update level to which they apply, according to the format:

```
me.VV.RR.LLLL
```

In this format the symbols VV, RR and LLLL represent the version, release and level numbers. You can have book change files for one or more of the program levels being updated.

4. One file named **./usr/sys/inst_updt/special** that identifies special update requirements for each program being updated. See "Creating a Special Requirement File" on page 9-36 for information about this file.

5. Program data for each program being updated, consisting of the following:

 - A library, created with the **ar** command and named **./usr/lpp/***pgm-name***/inst_updt/arp**, that contains the following files:

 update
 An executable file that contains a procedure to update the program. The procedure may be either a shell procedure, or a compiled procedure in **a.out** format.

 config
 An executable file that contains a procedure to update the system configuration. If the program does not do any customization, this file is not needed. The procedure may be either a shell procedure, or a compiled procedure in **a.out** format.

 al_*VV.RR.LLLL*
 An apply list file that contains the relative path names with respect to the / directory of all files to be updated. See "Creating an Apply List File" on page 9-34 for information about the file format. Include one apply list file for each update to the program. The symbols VV, RR, and LLLL represent version, release and level numbers as described in"Creating the Program History File" on page 9-26 .

 lpp.acf
 An optional file that defines archiving procedures for the program as described in "Creating an Archive Control File" on page 9-35.

 copyright
 An optional file that contains appropriate copyright information for this program. If this file is present, **updatep** displays the contents of this file when it applies the update to this program.

 *pgm-name*_**ver**
 An optional one-line file that contains the version information for a program. If this file exists, **updatep** use the first characters (up to 35) of its content as a comment record in the history file.

 fix*pgm-name*
 An optional file that contains the cumulative fix information for a program. If this file exists, **updatep** places it in the **/usr/lpp/***pgm-name***/apars** directory. It is then displayed to standard output when you enter the **updatep -A** command.

 - The new and replacement program files.

In addition, be sure to provide instructions for the update in the **instr** file (optional), instead of providing a manual that tells the operator how to do procedures required to update the program. The **instr** file should supply information needed for special configurations.

Using the updatep Program

In the library **./usr/lpp/***pgm-name***/inst_updt/arp**, provide a program, named **update** to do the **update** procedures that the program needs. The **updatep** program executes this program during the update procedure after it does the following preliminary tasks:

1. Restores the library that contains the update control files.
2. Determines which files need to be updated.
3. Extracts the update procedure **update** from the library file.
4. Starts the update procedure.

When the update procedure completes, the **updatep** program does the following cleanup tasks:

1. Updates the **lpp.hist** file to the indicated revision level.
2. Performs an IPL, kernel rebuild or shutdown based on the code returned by the update procedure.

Sending a Return Code to updatep

When the **update** program completes, it must send a return code to the **updatep** program. This return code signals the end of the **update** program, and determines what actions the **updatep** program will do next. Use one of the following values for a return code:

Code	Description
0	Successful completion. No additional action is needed.
2	Successful completion. The **updatep** program updates superblocks and the i-node list, and flushes the buffers (**sync**). Then it performs an IPL on the operating system.

Figure 9-4 (Part 1 of 2). Return Codes to updatep

Code	Description
3	Successful completion. The **updatep** program uses the **cfgaply** routine to build a new kernel. Then it updates superblocks and the i-node list, and flushes the buffers. Then it instructs the user to IPL the system and shuts down the operating system.
4	Successful completion. The **updatep** program uses the **cfgaply** routine to build a new kernel. Then it updates superblocks and the i-node list, and flushes the buffers before it performs an IPL on the operating system.
5	Update cancelled by the **update** procedure without errors.
6	Successful completion. The **updatep** program updates superblocks and the i-node list, and flushes the buffers (**sync**). Then it instructs the user to IPL the system and shuts down the operating system.
7	The update was cancelled by the **update** procedure with errors. The **updatep** program recovers the previous state of the system.

Figure 9-4 (Part 2 of 2). Return Codes to updatep

Example Update Procedure

Figure 9-5 on page 9-20 shows a simple example shell procedure to update an example program. The example procedure is simple. It would be more complicated if the update included a kernelupdate and required an IPL following the update. In this program the parameters have the following meaning:

$1	A variable that passes the full path name of the apply list file to the update procedure.
$2	A variable that passes the device path the default is **/dev/rfd0**.
programname	The name of the program (up to 8 characters)

```
programname = pgm-name
/etc/inusave  $1  "$programname"
rc=$?
if test $rc -ne 0
then
    exit  $rc          # return an error code
fi

/etc/inurest -d"$2" $1 "$programname"
rc=$?
if test $rc -ne 0
then
   exit $rc
fi

# The following section exits with a return
# code of 4 (kernel rebuild) if file myfile.o
# was included in the apply list.

fgrep myfile.o $1
rc=$?
if test $rc -eq 0
then
   exit 4
fi

exit 0
```

Figure 9-5. Example of update Procedure Code

Installing and Updating Active Files

When using **installp** and **updatep** with files on the system that are active at the start of the installation, the **instal** or **update** procedure provided with the program should ensure that the files are not active during the installation. The following suggestions can help to ensure that active files are installed correctly:

1. Identify all files in your installation or update that could be active at the start of the procedure, or that may become active during the procedure. The following files are among those that are active even in a quiesced state:

 - **/bin/sh**
 - **/etc/cron**
 - **/etc/qdaemon**
 - **/etc/init**
 - **/usr/lib/errdemon**
 - **/etc/writesrv**
 - **/etc/updatep**
 - **/etc/inudatep**
 - **/etc/inudatep**
 - **/bin/tsh**
 - **/etc/auditbin**
 - **/etc/watch**
 - **/etc/auditpr**

 If you do not have any potentially active files, ignore the rest of these guidelines. Do not perform any of this processing on active files that are not included in the apply list.

2. In the **instal** or **update** procedure, provide code that performs the following functions in the indicated order:

 a. Call **inusave** as part of the normal procedure.

 b. First delete and then create a temporary directory named **inst_updt.actv** for each file system that is being updated. Each directory must be at the top level of the file system. For example, for active files in the **/usr** file system, the temporary directory must be **/usr/inst_updt.actv**. From a shell procedure, use the **rm -rf** command to delete the directory to prevent messages from being displayed. If an error occurs, return from the procedure with an error code of 40.

 c. *Move* all potentially active files in each file system to the temporary directory for that file system and rename the files in the format of `active.`*n*, where *n* is an integer or other unique identifier for each file. If an error occurs, return from the procedure with an error code of 50.

d. After moving each file, make an entry in the file **/usr/lpp/***pgm-name***/inst_updt.save/active.list**. (*pgm-name* represents the name of the program being installed.) The format of an entry in this file is:

```
active.n  pathname  dirname
```

Where:

n is an integer that identifies the file in the directory.
pathname is the full path name of the active file before being moved.
dirname is the full path name of the temporary directory.

For example, if the first active file is **/etc/init**, the entry in **active.list** is:

```
active.1  /etc/init  /inst_updt.actv
```

e. *Copy* the moved files from the active directory back to their original positions and relink any files that were linked to them before the move.

f. Call **inurest** to restore the updated versions of the files.

g. When the procedure completes successfully, return with one of the following codes:

2	If an operating system IPL is required
3	If the kernel must be rebuilt and a shutdown is required
4	If the kernel must be rebuilt and an operating system IPL is required
5	If the **updatep** process cancels the update without an error occurring
6	If a shutdown is required
7	If the **updatep** process cancels the update with errors.

For updates, ensure that the **special** file indicates that an IPL is required. If active files have been changed, return a code that causes **updatep** to IPL the system.

For installations, tell the user in the written instructions that the installation performs an IPL when it completes.

h. If **inurest** does not complete successfully, your procedure should:

- If installing, call **inurecv** to recover the previous version (including any active files), log the error condition in the history file, and return the **inurest** return code to **installp**.
- If updating, return the **inurest** return code to **updatep**. **updatep** then calls **inurecv** to recover the previous version (including any active files).

Installing and Updating /bin/sh or /etc/restore

The files **/bin/sh**, **/bin/tsh**, **/etc/inurest** and **/etc/restore** are special case active files. Use the following guidelines to install or update these files:

1. The user moves the file to a different name and copies it back to the original location in case of errors.
2. The user restores the file by name from the restore diskette.

 Note: The user must use the different name created in if the file being changed is **/etc/restore**.
3. Delete the entry for the file from the apply list.
4. If **/bin/sh** was restored, IPL the operating system.
5. The user executes the **updatep** or **installp** program.

Use an update instruction to tell the user how to perform the update, and how to recover from any errors. If the update fails or is rejected, the user must move the saved copies of these files back to their original positions. If **/bin/sh** is recovered, IPL the operating system.

Allowing for Recovery

The **update** program requires that you save enough information to allow it to recover the previous level of the program being updated. For simple updates you can save this information from your update procedure by calling **inusave** before trying to change anything. The **inusave** program saves the current version of any normal files, and can also save any constituent file in the directory **/usr/lpp/***pgm-name***/inst_updt.save** if you request that they be saved.

The **update** program either restores or deletes the saved files, depending upon how the update turns out. It deletes the saved files if:

- The user commits (accepts) the update
- An error occurs during the update process, but before any files have been changed by the update process.

It recovers the saved files if:

- The user rejects the update
- An error occurs during the update process after files have been changed by the update process.

If an error occurs after the update process has already changed any file, the **update** program does *not* delete the saved files after recovering them. In this case, the person installing the update must delete both the files in the **inst_updt.save** directory *and* the directory. Otherwise, a later install or update for that program will detect the presence of that directory, causing the installation to fail.

Recovering from Kernel or Configuration Changes

When your program makes changes to either the kernel or configuration files, you must provide additional information and take extra steps. Supply an update instruction to inform the user that the kernel or a particular configuration file is being changed. Include in the instruction a warning that the files involved in the changes must not be further changed until the user either commits or rejects the update.

In addition, your update procedure must include instructions to perform the following steps controlled by the information contained in the apply list file, **al**:

1. Copy the current level of the file(s) being changed (either the kernel or the configuration file) into a file named,

 `/usr/lpp/`*`pgm-name`*`/inst_updt.save/config.`*`n`*

 In this name, the letter *n* is a unique number assigned by the procedure.

2. Add an entry in the file,

 /usr/lpp/*pgm-name*/inst_updt.save/config.list

 to document the saved file. Your procedure may need to create **config.list** if it does not already exist.

3. Run **inusave**.

4. Run **inurest**.

5. Perform any configuration dependent operations.

6. Return one of the following exit codes to **inuupdt**:

2	Sync and reboot
3	Use the **cfgaply** command to build the kernel, then sync and shutdown.
4	Use **cfgaply** to build the kernel. Then sync and reboot.
6	Sync and shutdown.

By following the preceding procedure, your program creates a copy of the current level of the updated file and provides information about whether a reboot is required to recover the current level. The **update** program can then recover from a rejected update.

Sequencing Configuration File Updates

When a program changes a particular configuration file more than once over a period of several updates, the changes must always be applied in the same order until the next release of the program. Your program's update procedure must adopt a file naming convention to allow it to do the updates in the required order.

For example, if a program contains changes for a configuration file called /etc/config.1 in its updates numbered 01.00.0010 and 01.00.0030, the updates could be contained in files named:

/usr/lpp/*pgm-name*/config.1/f.1
/usr/lpp/*pgm-name*/config.1/f.2

If the program being updated is at level 0000, the two changes could be applied in order. Similarly, if the program being updated is at level 0020, only the level 0030 change file would be included in the update apply list.

Creating the Program History File

The program history file **/usr/lpp/***pgm-name***/lpp.hist** contains information to identify the installed release and version of a program on the system. This file is an ASCII file. List these files to see samples of history files that exist on the system. See also *AIX Operating System Technical Reference* for more information about the history file.

Figure 9-6 shows the record format of the history file. Figure 9-7 defines the fields in the history record.

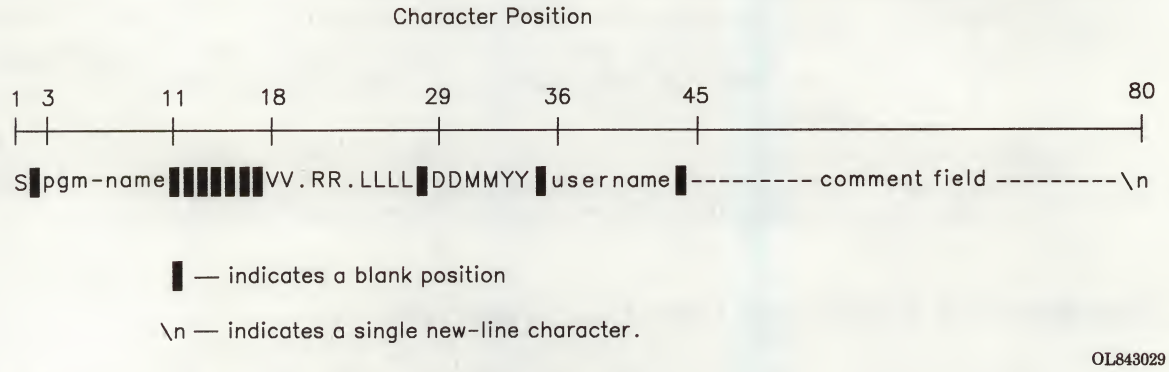


Figure 9-6. Record Format for the History File

Field	Description
S	Indicates the condition that the program is in, using the following characters: **a** An update was applied. **b** Positions 3-16 contain the name of the backup format file used to install this program. **c** An update was committed (accepted). **e** An unrecoverable error has occurred. **f** Positions 3-10 contain the 8-character name associated with this program subset. **m** A manual reject was performed. The state of the program is unknown. **p** Positions 3-10 contain the 8-character name and positions 18-27 contain the ver/rel/lvl, of the program subset associated with this program. **r** An update was rejected or recovered. **t** This record is a title record. Positions 3 through 32 contain a title for the program. **v** The VRM minidisk has been changed. This entry occurs only in the VRM history file (refer to *VRM Device Support* for more information). ***** This record is a comment field (put an * in position 79 to ensure a full length record).
pgm-name	The name assigned to the program (lowercase alphabetic characters only). If the name is less than 8 characters, the field must be filled out with blanks.
VV.	A 2-digit numeric field followed by a period indicating the version level of the program. The version number indicates which level of the hardware and operating system the program works with.
RR.	A 2-digit numeric field followed by a period indicating the release number of the program. The release number tracks changes to external programming interfaces since the last version change. This number increments each time the external interface to the program changes.

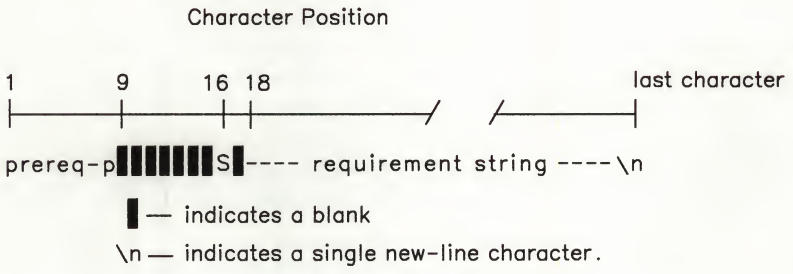
Figure 9-7 (Part 1 of 2). Fields in a History Record

Field	Description
LLLL	A 4-digit numeric field indicating the update state of the program. This field increments when the program changes and the change does not affect external programs that may use the documented external interface for the program. The level, together with the S field, ensures that all changes up to and including the current change are installed on the system. Change only the high-order 3 digits; the low-order digit is used for local changes.
DDMMYY	These three numeric fields indicate the date of the change to the program: DD Day of the month (01 to 31) MM Month of the year (01 to 12) YY Year (00 to 99)
username	An alphanumeric field that contains the user ID of the person that installed the program. If the user ID is less than 8 characters, this field must be filled out with blanks. This field is filled in at installation time, and can be blank when the update is distributed.
comment field	A 35-character field for adding comments.
\n	A required new-line character.

Figure 9-7 (Part 2 of 2). Fields in a History Record

Creating the Program Requirements File

You can use the **ckprereq** routine to determine if required programs are already installed on the system. The **ckprereq** routine uses the program requirements file **prereq** to make that determination. This file is an ASCII file. Figure 9-8 shows the record format of an entry in the requirements file. Figure 9-9 defines the fields in the requirements record.



OL843027

Figure 9-8. Record Format for Requirements File

Field	Description
prereq-p	An 8-character alphabetic field that contains the pgm-name of the program that is required.

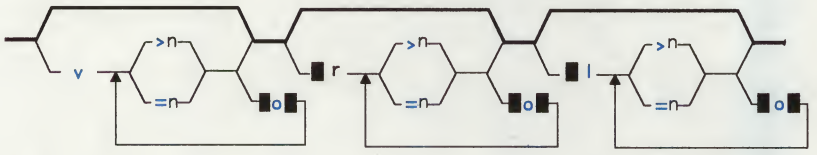
Figure 9-9 (Part 1 of 2). Fields in Requirements Record

Field	Description
S	**ckprereq** result code: Leave this field blank. The **ckprereq** command fills in this 1-character field with one of the following letters: **f** The format of the history file is not 80-byte records. **l** The requested level is not installed on the system. **n** A history file for that program is not installed on the system. **r** The requested release is not installed on the system. **s** The **prereq** file entry has a syntax error. **u** The state of the program is unknown. **v** The requested version is not installed on the system. **blank** The prerequisite program is installed on the system in the proper configuration.
requirement string	A set of logical expressions that define the version, release and level parameters that the prerequisite program must have. "Using the Requirement String" explains how to use the requirement string.
\n	Required new-line character

Figure 9-9 (Part 2 of 2). Fields in Requirements Record

Using the Requirement String

The requirement string allows you to specify a minimum version, release and level for the specified program. If the copy of the specified program does not meet the specified criteria, then the program to be installed should not be installed. The syntax of the requirement string and the symbols used to create the string are as follows:



■ indicates required blank
n can be any integer

OL843037

The symbols in the diagram have the following meanings:

v	A version number is being specified.
r	A release number is being specified.
l	A level number is being specified.
>	Greater than.
=	Equal.
o	Logical OR.
n	An integer value for the specific parameter to be compared against.

The **chkprereq** routine evaluates each requirement string in the **prereq** file beginning with the `v` (for version) and continuing to the right (left to right order). It does not evaluate any further than is required to determine that the whole line must be true or false. For example, if the program `myprog` is at version 3.1 (version 3, release 1), and the requirement string is:

```
myprog          v>1 r>0
```

The routine needs to determine only that version 3 is much larger than 1 to be sure that the correct conditions exist. If, however, `myprog` is at version 2.1, the routine must not only evaluate the version but also the release. Version 2.0 would fail the test; version 2.1 passes the test.

Requirements File Example Entry

The following entry:

```
myprog          v=2 r>30 o =10 o =15
```

in a requirements file indicates that:

- The program `myprog` is a required program.
- It must be version 2.
- Any level is valid (*l* is not specified).
- The release may be either 10, 15 or greater than 30.

Similarly, the following entry:

```
myprog          v>1 r>30 o =10 o =15
```

in a requirements file indicates that:

- The program `myprog` is a required program.
- It must be version 2 or greater.
- Any level is valid (*l* is not specified).

- If the version is 2, the release may be either 10, 15 or greater than 30; if the version is 3 or more, any release is valid.

Creating the Program Name File

The program name file contains the names of the programs on the set of installation or update diskettes. Name this file **./lpp_name**. It contains one or more entries.

Figure 9-10 shows the format of an entry in the program name file. Figure 9-11 defines the fields in the entry.

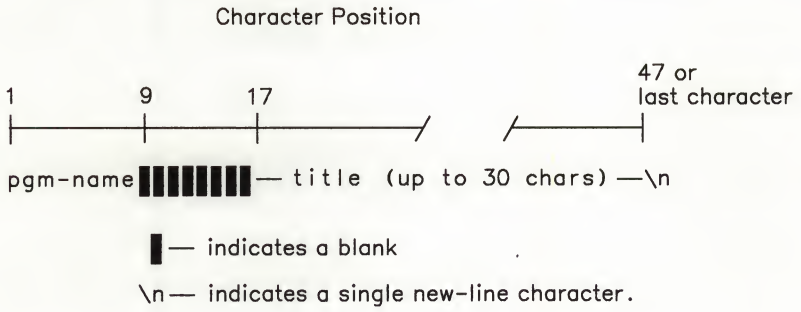


Figure 9-10. Entry Format for Program Name File

Field	Description
pgm-name	The name assigned to the program (lowercase alphabetic characters only). If the name is less than 8 characters, this field must be filled out with blanks.
title	A descriptive title of up to 30 characters.
\n	A required new-line character.

Figure 9-11. Fields in Program Name Entry

Creating an Apply List File

The apply list file contains an entry for each file to restore during an installation or an update procedure. The apply list file for an install, or the merged apply list for an update cannot be larger than 14K bytes. If the list is larger, then the change should not be an update. Use the install procedure to replace the program. Your **instal** program should then use the **restore** command, rather than **inurest** to restore the files.

Each entry in the file is a relative path name. The **installp** and **updatep** programs set the current directory to / before doing the restore operations. Therefore, the path names in this file specify the full installed path name of the file relative to /. For example, if the apply list file contains the following entry:

```
./usr/bin/newcmd
```

the program is installed as **/usr/bin/newcmd** in the system. Apply lists in program subset backup format files must conform to the following name conventions:

- More than one apply list can be provided
- Each apply list must be in the form **cp_al.***xxx* where:
 - **cp_al.** must be as shown
 - *xxx* is any **ASCII** string conforming to standard file name requirements.

In the process of creating a program subset backup format file, the apply lists in the library are concatenated and then uniquely sorted to ensure that all applicable files are included.

Creating an Archive Control File

The **install** and **update** programs use the archive control file to replace individual member files into libraries that other programs own, such as adding a new file to the kernel. The **update** program also uses the archive control file to replace individual member files in libraries that it owns.

Supply an ***archive control file*** in the library **liblpp.a** for an install or in the **arp** file for an update (if the archive control file is not already present, or if the archive control file needs to be changed for the update procedure). The archive control file allows you to:

- Save a library member file
- Archive restored files into a library

during an installation or update procedure. The archive control file lists the library member files to save along with the name of the library file where the file can be found. Name the archive control file **lpp.acf**. This file consists of one or more lines. Each line has the following format:

filename *archive_file*

The parameters on each line are separated by one or more blanks, and have the following meaning:

`filename` The relative full path name (with respect to /) where the library member file should be restored.

`archive_file` The full path name of the library file that contains the member file **filename**.

When **lpp.acf** is present, the installation program can use the **inusave** command to save the listed files before it installs an update to the files. The **inusave** command compares the **filename** parameters in this file with the files to be updated (listed in the apply list file). If the same filename appears in both places, the **inusave** program saves the old version of the file before copying the new version of the file into the system. The **inusave** program does not produce an error message if a file listed in the apply list file does not exist. The **inurest** command also uses **lpp.acf** to archive the listed files.

Note: You should not store your files in any of the system libraries (listed on page Figure 3-1 on page 3-4) because you could lose the files if you reinstall or update the AIX Operating System. You should keep your files in your own separate library.

Creating a Special Requirement File

The **updatep** command uses the special requirement file to determine the grouping of descriptive titles to be displayed to the user during the update procedure. You must provide this file, even if it is zero length (empty). Name this file **./usr/sys/inst_updt/special**. Depending on the program and the update being applied, this file can contain many types of entries, as described in the following paragraphs.

Figure 9-12 defines the variable parameters used in the following paragraphs.

Entry	Definition
`pgm-name`	The name of the program.
`other-pgm`	The name of another program that this program depends on.
`LLLL`	The level of the program that causes the change.
`\n`	Required new-line character.

Figure 9-12. Requirements File Parameters

Note: If **updatep -ac** (apply and commit) is specified, you can select all special requirement for update at the same time. This allows for multiple program updates and requires only one kernel rebuild or system IPL. If **update -a** (apply only) is specified, special requirement programs must be updated one at a time.

Dependent Program Entry

The dependent program entry indicates that this update includes changes for another program that must also be applied. The format of this entry in the special requirement file is:

`coreq` *pgm-name LLLL other-pgm*`\n`

If the special requirements file contains another entry for either of the programs listed in the dependent program entry, then both programs must be updated by themselves. They cannot be updated together as dependent programs. Include information in your update instructions to tell the user to update both programs independently.

Changes to Report Templates Entry

This entry indicates that the update includes changes to the report templates for error tracking or trace reports. These templates are in the files **/etc/errfmt** and **/etc/trcfmt**. When this entry is present, the update for the program must be installed and accepted independent from updates for other programs. When using this entry, use an update instruction in the **instr** file to tell the user to recover the template files that were saved during the update procedure. The format of this entry in the special requirement file is:

`ras` *pgm-name* *LLLL*`\n`

IPL Required Entry

This entry indicates that the operating system must be loaded and started again following installation of this update. When this entry is present, the update for the program must be installed and accepted independent from updates for other programs. The format of this entry in the special requirement file is:

`ipl` *pgm-name* *LLLL*`\n`

Changes to Configuration Entry

This entry indicates that the update includes a change to the system configuration. When this entry is present, the update for the program must be installed and accepted independent from updates for other programs. The format of this entry in the special requirement file is:

`config` *pgm-name* *LLLL*`\n`

Changes to Kernel Rebuild Entry

This entry indicates that the update rebuilds the kernel. when this entry is present, the update for the program must be installed and accepted independently from updates for other programs. The format of this entry in the special requirement file is:

`ker` *pgm-name LLLL*`\n`

Changes to VRM Entry

This entry indicates that the update includes a change to the VRM. When this entry is present, the update for the program must be installed and accepted independently from updates for other programs. The format of this entry in the special requirement file is:

`vrm` *pgm-name* *LLLL*`\n`

Creating a Save and Recover Directory

The save and recover directory (**/usr/lpp/***pgm-name***/inst_updt.save**) contains copied files and extracted archived files that were saved during a reinstallation, or during application of an update. If a reinstallation procedure created the directory, the installation procedure must run **inurecv** if an error occurs, or delete the directory when installation is complete. If an update procedure created the directory, the directory exists until the update is either committed or rejected, or until **updatep** recovers the directory because of an error during the update procedure. At that time the **updatep** program deletes this directory, after recovering the saved files if the update was rejected. However, if the update is rejected without automatic recovery, the directory is not deleted to allow for manual recovery of the saved files. The directory must then be manually deleted after the files are recovered.

The directory contains the files listed in Figure 9-13. The order is not important. The directory can contain only the file names in the list.

File	Contents
update.list	An optional file that lists any regular files that were saved in this directory as described by **update.*n*** below. This file consists of one record for each file saved. This record has the following format: `update.`*n* *pathname*`\n` Where *pathname* is the full path name of the file when it is restored on the system, and `\n` represents a single new-line character.
update.*n*	A file named in this form for each record in the **update.list** file. In this form for naming the file: **update** Identifies the file as a backed up file. **n** Is an integer. Each backed up file has a unique *n* associated with it, starting with 1 for the first file backed up. For example, three backed up files have the names `update.1`, `update.2` and `update.3`.

Figure 9-13 (Part 1 of 4). Save/Restore Directory Content

File	Contents
archive.list	An optional file that lists any archived files that were extracted and saved in this directory as described in **archive.n** below. This file consists of one record for each archive file saved. This record has the following format: `archive.`*n* *member-name archive-name*`\n` Where *member-name* is the full path name of the file when it is restored on the system, *archive-name* is the full path name of the target archive file where this file belongs, and `\n` represents a single new-line character.
archive.*n*	A file named in this form for each record in the **archive.list** file. The file contains the file that was saved as listed in the **archive.list** file. In this form for naming the file: **archive** Identifies the file as a backed up archive file. **n** Is an integer. Each backed up file has a unique *n* associated with it, starting with 1 for the first file backed up. For example, three backed up files have the names `archive.1`, `archive.2` and `archive.3`.
config.list	An optional file that lists any configuration files that were saved in this directory as described in **config.n** below. This file consists of one record for each configuration file saved. This record has the following format: `config.`*n* *pathname*`\n` Where *pathname* is the full path name of the configuration file when it is restored on the system, and `\n` represents a single new-line character.

Figure 9-13 (Part 2 of 4). Save/Restore Directory Content

File	Contents
config.*n*	A file named in this form for every record in the **config.list** file. The file contains the file that was saved as listed in the **config.list** file. In this form for naming the file: **config** Identifies the file as a backed up configuration file. **n** Is an integer. Each backed up file has a unique *n* associated with it, starting with 1 for the first file backed up. For example, three backed up files have the names `config.1`, `config.2` and `config.3`.
active.list	An optional file that lists any active files that were saved by the program procedure. This file consists of one record for each active file saved. Each record has the format: `active.`*n filename directory*`\n` In this form the entries mean: *n* An integer that identifies the active file. *filename* The full path name of the active file before it was saved in the indicated directory *directory* The full path name of the directory where the active file is saved `\n` A single new-line character
active.*n*	A file named in this form for every record in the **active.list** file. The file contains the file that was saved as listed in the **active.list** file. In this form for naming the file: **active** Identifies the file as a backed up active file. **n** Is an integer. Each backed up file has a unique *n* associated with it, starting with 1 for the first file backed up. For example, three backed up files have the names `active.1`, `active.2` and `active.3`.

Figure 9-13 (Part 3 of 4). Save/Restore Directory Content

File	Contents
uniq_dir.list	An optional file that lists any install or update directories that were created during the process of archiving library members during an update. It permits **updatep** to delete any **inst_updt**/*libname* directories when committing or rejecting an update (except for the directories **/usr/sys/inst_updt** and **/usr/lpp/***pgm-name***/inst_updt** which are used by the update process). The file consists of one record for each member file in the directory that is created. The record consists of the full path name of the member file: */path*/inst_updt/*libname/member* In this form the parts have the following meanings: *path* The full path name to the **inst_updt** directory *libname* The name of the library that was created *member* The name of the member file

Figure 9-13 (Part 4 of 4). Save/Restore Directory Content

Designing Programs for a Code-Service Environment

A code-service environment is a network of client machines and server machines. The server machines can have many program products installed; the client machines can have very few. The clients have access to, and run, programs installed on the servers as if they were installed locally.

A code-service environment, in many cases, can provide the following benefits:

- The program products are only installed once, at the server, saving disk space on all of the client machines.
- Installation of new programs and updates to installed programs need only be done at the server, saving this administrative work at all of the client machines.
- All machines in the client-server network use programs that are at a compatible level.

Each machine in a client-server network must have at least the following programs installed:

- The AIX Operating System
- SNA
- Distributed Services
- Either a token-ring device driver or a baseband device driver.

To gain access to a server, the client mounts the following server directories over its own directories of the same name:

- **/usr/bin**
- **/usr/include**
- **/usr/lib**
- **/usr/lpp**

Once these server directories are mounted over the client directories, the client sees the same program products, **include** files, and libraries as the server. This is how the code service is provided.

Program products should be designed as if the previously listed directories are the only ones mounted over. However, for historical reasons the following server directories are also mounted over the client directories:

/usr/datamgt **/usr/database**
/usr/dos/bin **/usr/man**
/usr/pub **/usr/games**

Code-Service Types

Most program products can be designed to fit smoothly into a code-service environment and provide all of the expected benefits; these are called ***server-exclusive*** installations and usually result in the ***complete program*** being installed at the server.

Other programs require some type of installation activity at each client, and thus do not provide all of the benefits; these are called ***client-partial*** programs and usually result in part of the program (a ***program subset***) being installed at the client.

Server-Exclusive Programs

Server-exclusive programs are installed, and updated, exclusively at the server. No actions are required at the client. In order to design such a program, you should consider the following:

- The program install operation should only place files in or below the mounted-over directories. Files should not be placed in directories such as **/etc, /lib,** or **/usr/sys** because client systems do not have access to these server files and thus behave differently than stand-alone machines.
- Program configuration should be separate from installation. For example, the install procedure should not test the hardware configuration of the target machine and then use the result of the test to decide which subset of the program product code to install. If this is done, clients with different hardware configurations do not run properly.

 If this kind of node-unique configuration is required, have a separate configuration program run by an administrator at each client. Such node-unique configuration data should be stored in or below **/etc**; not in or below **/usr/lpp/***pgm-name* where it cannot be node-unique.

 There might be some simple changes that need to be made to files or directories at the client that are not mounted over by the server. For example:

 - Modifications might be required to **/etc/rc.include**, for instance.
 - Directories might need to be created for spool files in or below **/usr/spool**.

 You can cause such simple changes to be performed at clients in the following ways:

 - You can put a **README** file in the directory **/usr/lpp/***pgm-name* that instructs the client administrator to make simple changes.
 - You can put a program in or below **/usr/lpp/***pgm-name* that the client administrator is instructed to run.
 - You can describe the needed changes in the program documentation.

 These methods violate the definition of a server-exclusive program product. However, if the client-unique actions are simple and are not likely to be modified by future

updates, these methods are preferable to installing program subsets at the clients. For more information see "Client-Partial Programs" on page 9-46

- If the program has some user customization (such as **/$HOME/.profile**), the defaults for this customization should be placed in or below **/usr/lpp/***lppname* rather than in a directory that is not overmounted. The user should copy the defaults to **/$HOME** (or another appropriate directory) and personalize them. This makes the customization user unique, and the defaults system unique; there is no need for node-unique values.
- The server mounts over directories are read-only. Therefore, the program should not attempt to alter files in or below these directories.
- Directories not reached through one of the mounted over directories are not visible to all of the clients and therefore your program cannot use **/etc/locks** to synchronize access to data; use file locks (**fcntl**) instead.
- Distributed Services does not support mapping of remote files. Therefore, a program product could fail if it cannot map a file that resides in or below one of the mounted over directories. In some instances, use of mapped files gives better performance in a stand-alone machine than accessing a non-mapped file via file system calls. Since the file system calls continue to work on a mapped file, and they maintain the performance advantage of mapping, programs can maintain a performance advantage when running in a stand-alone machine without losing the advantages of being a server-exclusive product by:
 - Writing the program so it uses file system calls to access a file rather than memory load and store operations.
 - Mapping the file, but treating a failure (because the file is remote) as a non-fatal error.

 In some instances use of mapped files makes a program simpler because you can use array indexes and pointers rather than read and write system calls. In these cases, if the file is remote the data can be:

 1. Read from the file into a shared memory segment
 2. Operated on in memory
 3. Written back to the file.

 This is not remote/local transparency, but the programming method can be the same for remote and local files.
- In a code service network the same physical copy of the program is run, perhaps concurrently, by several different processors. Licenses, terms and conditions, and copy protection schemes must allow this. Some products use a password scheme based on the processor read-only-storage identifier (**uname -m**) to ensure that one copy of the program runs on only a specific processor. These schemes should be expanded to allow (with proper terms and licenses) one copy of the program to run on any of a list of processors.

Client-Partial Programs

Programs that alter or add files that are not in the server directories that mount over the client are called client-partial programs. These include programs that install device drivers because they must make entries in one or more of the following client files or directories:

- **/etc/ddi**
- **/etc/vrmdd**
- **/etc/system**
- **/etc/master**
- **/usr/sys/lib2**

These programs might also cause the operating system to be rebuilt.

A client-partial program needs two different sets of installation data:

- The files and procedures to install the program onto a stand-alone machine or onto a code server. This is the complete program installation data.
- The files and procedures to install the client-unique portions of the program onto a client machine. Typically, this is a subset of the complete installation procedure that results in a subset of the complete program being installed on client machines; this is the client-partial (program subset) installation data. Both the complete program and the program subset installation data can be on the same distribution media.

The install utility **bffcreate** reads the distribution media and creates two backup format files: one for complete program installations, and one for program subset installations. The backup format file for the program subset installation appears to **installp** as if it is data for an independent program product. **installp** is not aware of the relationship between the program subset installation data and the complete program installation data.

To create the backup format file for a program subset installation, **bffcreate** uses the information in the library file with the reserved name **./usr/lpp/***pgm-name***/liblpp.cp.a**. The primary elements in this library are:

- **lpp.hist** — This file contains program subset installation history, and also the name the client-partial (program subset) is to be known by.
- **cp_al.***xxx* — This file contains the apply list for the program subset installation data. This specifies which files are to be copied from the distribution media to the program subset installation data backup format file.

bffcreate performs the following:

1. Extracts **lpp.hist** and **cp_al.***xxx* from the **./usr/lpp/***pgm-name***liblpp.cp.a** library.
2. Reads **lpp.hist** to determine the **cp_***pgm-name* the program subset is to be known by. **bffcreate** creates a **lppname** file, writes the program subset **cp_***pgm-name* into it, and arranges for this to be part of the installation data.
3. Renames **./usr/lpp/***pgm-name***/liblpp.cp.a** to **./usr/lpp/cp_***pgm-name***/liblpp.cp.a**.

4. Copies the files named in **cp_al.*xxx*** to the program subset installation data backup format file.

A simple scenario might be as follows:

- The server administrator acquires the four installation diskettes for a program product that has both complete program (server-exclusive) and program subset (client-partial) installation data. The name of the program product is `fancypgm`, and the name chosen for the subset is `cp_fancy`.
- The server administrator removes the server from the network by running a **stop sna** command. This is normally done when clients are not apt to be using the server.
- The server administrator installs `fancypgm` on the server by running the following command:

  ```
  installp -b
  ```

 The **-b** flag causes **installp** to run the **bffcreate** utility. This utility reads the four diskettes as input and creates two files in backup format:

 - A copy of the program distribution diskettes (named `/usr/lpp.install/fancypgm.01.02` where `01` is the version and `02` is the release of the program).
 - A copy of only the program subset files included in the original distribution diskettes (named `/usr/lpp.install/cp_fancy.01.02`).

 After **bffcreate** finishes, **installp** completes the normal installation process except it uses the newly created program copy from file `/usr/lpp.install/fancypgm.01.02` as the distribution file instead of diskettes.

 The server is once again ready to support active service clients, and the server administrator places it back in the network with the **start sna** command.
- Each client attempts to re-attach to the server in active service mode when:
 - An initial program load (IPL) is performed which causes **chngstate** to run.
 - The client administrator issues the **chngstate** command.

 chngstate determines that a program installed at the server requires a program subset to be installed at the client. If **mode** in the client attribute file `/etc/codeserve/serverattach` is set to **auto**, then **chngstate** causes **installc** to install the program subset on the client machine. In this example, `cp_fancy` is installed across the distributed services network through access to `/usr/lpp.install/cp_fancy.01.02` on the server machine.

 After the program subset is installed on the client, the client attaches to the server in active service mode by mounting the ten code-server directories over its own

directories of the same names. The client then runs `fancypgm` as though it is installed locally.

Chapter 10. Maintaining Different Versions of a Program

CONTENTS

About This Chapter 10-2
Introducing SCCS 10-3
 Features 10-3
 New Terms 10-4
 SCCS File Format 10-5
 Command Conventions 10-6
 Command Summary 10-7
Using SCCS Commands 10-8
 Using the admin Command 10-9
 Using the get Command 10-11
 Using the delta Command 10-16
 Using the help Command 10-17

About This Chapter

This chapter shows how to use the Source Code Control System (SCCS) to control revisions to source code or documentation files using the major SCCS commands.

First it gives background information about SCCS. This includes new terms, the format of SCCS files, and how to use the SCCS commands.

Next it shows a sample SCCS session and describes the three major SCCS commands in detail. These commands are **admin**, **get**, and **delta**. The other commands are described in *AIX Operating System Commands Reference*.

Introducing SCCS

The ***Source Code Control System*** (SCCS) allows one user or a group to control and account for changes made to source code or documentation files. It stores the changes made to a file instead of storing the changed file. This allows several versions of the same file to exist in the system. To edit the file, specify the version. SCCS builds that version based on its stored information about previous changes made. Using SCCS reduces storage requirements and helps track the development of a project that requires keeping many versions of large programs.

Features

The SCCS commands form a complete system. Once you create an SCCS file, use an SCCS command to change it. Do not edit or compile the SCCS file itself. Use another file that is derived from the original SCCS file for these operations.

SCCS commands can do the following:

- Create an SCCS file
- Get a version of an SCCS file
- Save changes made to that file version
- Define who can change an SCCS file
- Record who made changes to the SCCS file
- Record when and why the changes were made.

New Terms

The following descriptions of SCCS terms are used in this chapter.

SCCS file Any file containing text (source code or documentation) that is controlled with SCCS commands. All SCCS files begin with `s.`. This file contains the original file contents and sets of changes to the original file or later versions of that file. It also contains information about who can change the file, who made changes and when they were made. Do not edit this file directly. It contains information to build the stored files.

delta A set of changes made to an SCCS file. After changing a file, use the **delta** command to save those changes in the SCCS file, thereby creating a new delta. Create a new delta only to save the changes made. When editing a specific version of an SCCS file, that version may consist of several different deltas.

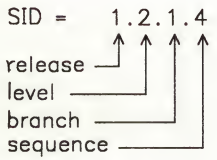
SID ***SCCS Identification***: The name assigned to a delta. An SID has up to four parts as shown in Figure 10-1 on page 10-5.

Every SCCS file starts out with an SID of 1.1. which means release 1, level 1. After editing version 1.1 and saving the changes, SCCS gives the new delta an SID of 1.2, which means release 1, level 2.

A typical SCCS file only uses release and level numbers and grows in a straight line. In these cases, the latest file version uses every previous delta to that file. However, a file may branch to a path where a file version consists of a subset of all of the deltas. For example, a common file can be used by two different groups. Both groups need the same code up to a certain point, and then each group goes its own way. In this case, create a branch delta that allows each group to add deltas onto a common base.

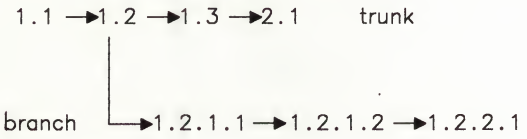
The file then has a trunk, with deltas identified by release and level, and one or more branches, which have deltas containing all four parts of an SID. On a branch, the release and level numbers are fixed and new deltas are identified by changing sequence numbers. Note that a file version built from a branch does not use any deltas placed on the trunk after the point of separation. See Figure 10-2 on page 10-5.

SCCS can combine different deltas. However, when combining many deltas into one, SCCS loses track of the changes that created each of the smaller (old) deltas and only tracks the change to create the larger (new) delta.



A5AC6009

Figure 10-1. Parts of an SID



A5AC6008

Figure 10-2. Growth of an SCCS File with Branching

SCCS File Format

SCCS files have two major sections called the header and the body. The header has five subsections, which identify who created the file, who can change it and other administrative details. The body has one or more subsections consisting of the text portions of the file. There is one text portion for each delta in the SCCS file.

The **get** command uses the header and body to create the specified file version. Other SCCS commands make use of the header and body to perform their functions.

Never edit the SCCS file itself except to modify incorrect information in the header as described in "Locating Damaged SCCS Files" on page 10-10. Editing the SCCS file can damage the structure of the file. The SCCS commands can do any necessary modifications of the header or body. To see how an SCCS file is organized, examine an SCCS file using simple commands such as **cat** or an editor. However, do not edit the SCCS file.

The SCCS File Header

The sections of an SCCS file header are described in the following.

Checksum A number containing the logical sum of all of the characters in the file. The **admin** command uses the checksum to ensure that all changes to the file were made by using the SCCS system.

Comments Descriptive text provided by the user to describe the contents or purpose of the file.

Delta Table Information about each delta including type, SID, date and time of creation, and comments.

User Names List of login names or group IDs of users who are allowed to modify the SCCS file by adding or removing deltas. If this parameter does not exist, anyone can modify the SCCS file.

Options List of indicators that control specific actions of various SCCS commands.

The SCCS File Body

The SCCS File body includes the actual text of the deltas in the file. The body also contains SCCS control text intermixed with the delta text. Notice that the deltas are in reverse order. That is, the most recently created delta is the first one in the list.

Warning: Using non-SCCS commands with SCCS files can damage the SCCS files. Changing an SCCS file with a non-SCCS command makes the checksum incorrect. See "Locating Damaged SCCS Files" on page 10-10 for information about working with the checksum.

Command Conventions

In most cases, SCCS commands accept the following two types of parameters:

flags Flags begin with the - (minus sign), followed by a lowercase character, and sometimes followed by a value. They control how the command operates.

arguments Arguments may be file or directory names. They specify the file or files with which the command operates. Using a directory name as an argument specifies all SCCS files in the directory.

Arguments cannot begin with a - (minus sign). If you specify the - (minus sign) by itself, the command reads standard input until it reaches an end of file character (Ctrl-D). This can be useful when using pipes. When using the keyboard for input, it reads until it finds **Ctrl-D**.

Any flags specified for a command apply to all files specified for that command and are processed before arguments to that command. Their placement in the command line is not important. Arguments are processed left to right. Some SCCS files contain flags that determine how the command operates on the file. See "Using the admin Command" on page 10-9 for more information.

SCCS Commands produce error messages with the following format:

```
ERROR [file]: message text (code)
```

The code in parentheses can be used as an argument to the **help** command. The **help** command can sometimes provide more information about a particular error code.

An SCCS command stops processing a file that contains a fatal error. Any other files in the command are still processed.

Command Summary

The following summary presents the commands in the order of their use. They are further defined in *AIX Operating System Commands Reference.*

admin	Creates an SCCS file or changes some characteristic of an existing SCCS file.
get	Gets a specified version of an SCCS file. Use this command to get a copy of a file to edit or compile.
unget	Undoes the effect of a previous use of the **get -e** command.
delta	Adds a set of changes (delta) to the text of an SCCS file.
rmdel	Removes a delta from an SCCS file. The delta must be the most recent delta on its branch.
cdc	Changes the comments associated with a delta.
what	Searches a system file for a pattern and displays what follows it. Use this command to find identifying information.
sccsdiff	Shows the differences between any two versions of an SCCS file.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta. Combining deltas may reduce storage requirements.
val	Checks an SCCS file to see if its computed checksum matches the figure listed in the header.
prs	Prints portions of an SCCS file in a specified format.
help	Provides an explanation of a diagnostic message.

Using SCCS Commands

Figure 10-3 shows how to create, change and update the contents of an SCCS file. The SCCS commands have much more function than what is shown in the figure. System responses to the SCCS commands are not shown.

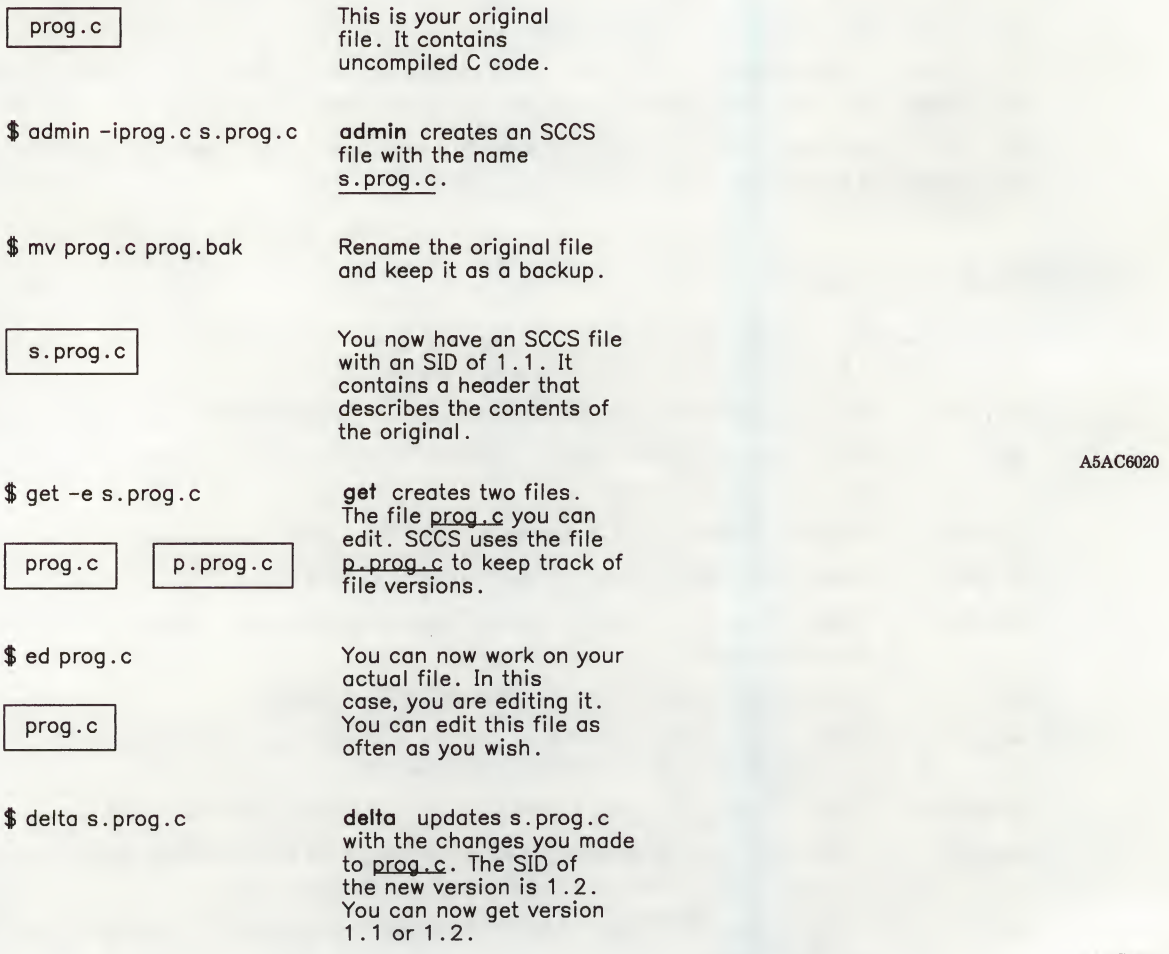


Figure 10-3. Example of Using SCCS to Create and Update a File

Using the admin Command

These examples use an imaginary text file called `test.c`, and an editor such as **ed** to edit files.

First, create an ordinary SCCS file. If you use the **-i** flag, **admin** creates delta 1.1 from the specified file. Without the **-i flag**, **admin** creates an empty SCCS file. Once delta 1.1 is created, rename the original text file so it does not interfere with SCCS commands. For example, to create an SCCS file from a file `test.c`:

```
$ admin -i test.c s.test.c
No id keywords (cm7)
$ li
s.test.c test.c
```

Then rename the original text file:

```
$ mv test.c  back.c
```

The message, `No id keywords (cm7)` does not indicate an error. SCCS writes this message when there are no identification keywords in the file. Identification keywords are variables that can be placed in an SCCS file. The values of these variables provide information, such as date, time, SID, or file name. See "Getting Read-Only File Versions" on page 10-11 for an explanation of identification keywords. If there are no identification keywords, SCCS writes the message.

Name the SCCS file anything as long as it begins with `s.`. In the preceding example, the original file and the SCCS file have the same name, but that is not necessary.

Because you did not specify a release number, **admin** gave the SCCS file an SID of 1.1. SCCS does not use the number 0 to identify deltas. Therefore, a file cannot have an SID of 1.0 or 2.1.1.0. All new releases start with level 1. To start the `test.c` file with a release number of 3, use the **-r** flag with the **admin** command, as shown below:

```
$ admin -i test.c -r3 s.test.c
```

To restrict permission to change SCCS files to a specific set of user IDs, list their user IDs or group ID numbers in the user list of the SCCS file by using the **-a** flag of the **admin** command. These IDs then appear in the SCCS file header. Without the **-a** flag to restrict access, all user IDs can change the SCCS files. The following example restricts permissions to the user ID `dan`:

```
$ admin -adan s.test.c
```

Locating Damaged SCCS Files

Although SCCS provides some error protection, you may need to recover a file that was accidentally damaged. This damage may result from a system malfunction, operator error, or changing an SCCS file without using SCCS commands.

SCCS commands use the checksum to determine whether a file was changed since it was last used. The only SCCS command that processes a damaged file is the **admin** command when used with the **-h** or **-z** flags. The **-h** flag tells **admin** to compare the checksum stored in the SCCS file header against the computed checksum. The **-z** flag tells **admin** to recompute the checksum and store it in the file header.

Check SCCS files on a regular basis for possible damage. The easiest way to do this is to run the **admin** command with the **-h** flag on all SCCS files or SCCS directories as shown below:

```
$ admin -h s.file1  s.file2 ...

$ admin -h directory1  directory2  ...
```

If **admin** finds a file where the computed checksum is not equal to the checksum listed in the SCCS file header, it displays this message:

```
corrupted file (co6)
```

If a file was damaged, try to edit the file again, or read a backup copy. After fixing the file, run the **admin** command with the **-z** flag and the repaired file name:

```
$ admin -z s.file1
```

This operation replaces the old checksum in the SCCS file header with a new checksum based on the repaired file contents. Other SCCS commands can now process the file.

Using the get Command

The **get** command gets files in either ***read-only*** or ***editable*** form. You can only use the ***editable*** form to create a delta to the SCCS file. Use the ***read-only*** form to print or compile the file. The results of using **get** on a file depend on whether you specify the file as read-only or editable. The examples are divided into read-only examples and editable examples. The examples are not related unless the comments specify that they are related.

Note: You must use the **-e** flag with the **get** command, to create a delta.

Getting Read-Only File Versions

To compile a program or print a document from an SCCS file, get the file as read-only. The **get** command performs different tasks when it gets a read-only file.

The difference between the two types of **get** operations is important when using identification keywords in a file. Identification keywords can appear anywhere in a file. They are symbols that are replaced with some text value when **get** retrieves the file as read-only. For example, to print the current date and SID in a file, put the following symbols in the file:

```
%H%  %I%
```

%H% is the symbol for the current date and %I% is the symbol for the SID. When **get** retrieves a file as editable, it leaves the symbols in the file and does not do text value substitution. See *AIX Operating System Commands Reference* for the identification keywords to use in a file.

Several examples of the **get** command are shown below:

```
$ li
s.test.c
$ get s.test.c
3.5
59 lines
$ li
s.test.c test.c
```

Because you did not specify a version of the file, **get** built the version with the highest SID. In the next two examples, the **-r** flag specifies which version to get:

```
$ get -r1.3 s.test.c
1.3
67 lines
$ get -r1.3.1.4 s.test.c
1.3.1.4
50 lines
```

If you specify just the release number of the SID, **get** finds the file with the highest level within that release number.

```
$ get -r2 s.test.c
2.7
21 lines
```

If the SID specified is greater than the highest existing SID, **get** gets the highest existing SID. If the SID specified is lower than the lowest existing SID, SCCS writes an error message. In the following example, release 7 is the highest existing release:

```
$ get -r9 s.test.c
7.6
400 lines
```

The **-t** flag gets the top version in a given release or level. The top version is the most recently created delta, independent of its location. In the next example, the highest existing delta in release 3 is 3.5, while the most recently created delta is 3.2.1.5.

```
$ get -t -r3 s.test.c
3.2.1.5
46 lines
```

Getting Editable File Versions

All of the previous examples use the **get** command to get a read-only file. To edit the file and create a new delta, use the **-e** flag. The **get** command works differently when using the **-e** flag, so the previous examples may not apply. Restrictions for files built with the **-e** flag are explained in the *AIX Operating System Commands Reference* under the **get** and **admin** commands. If you build the wrong version of the file, use **unget** to undo the effect of the **get -e** command.

Several examples of the **get** command are shown below:

```
$ li
s.test.c
$ get -e s.test.c
1.3
new delta 1.4
67 lines
$ li
p.test.c  s.test.c  test.c
```

The working file is `test.c`.. If you edit `test.c` and save the changes with the **delta** command, SCCS creates a new delta with an SID of 1.4. The file `p.test.c` is a temporary file used by SCCS to keep track of file versions.

In the previous example, you could use the **-r** flag to get a specific version. Assuming delta 1.3 already exists, the following three uses of the **get** command are the same:

```
$ get -e s.test.c
$ get -e -r1 s.test.c
$ get -e -r1.3 s.test.c
```

To start using a new (higher in value) release number, get the file with the **-r** flag and specify a release number greater than the highest existing release number. In the next example, release 2 does not yet exist:

```
$ get -e -r2 s.test.c
1.3
new delta 2.1
67 lines
```

Notice that **get** indicates the version of the new delta that will be created if the **delta** command stores changes to the SCCS file. If the example did not include the **-e** flag, **get** would build the highest existing SID (1.3) and would not indicate a new delta, even though the **-r2** flag requests a version 2.1.

To develop a version of the SCCS file that does not depend on the most recently created delta, create a branch delta as shown in Figure 10-4.

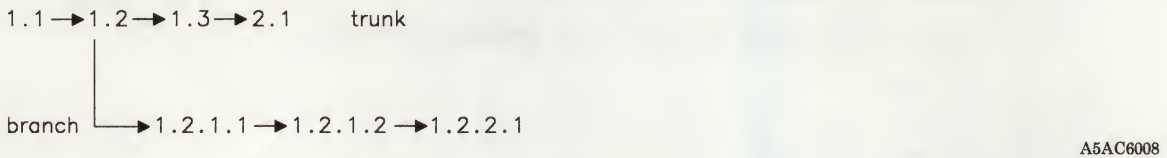


Figure 10-4. Growth of an SCCS File with Branching

In the figure, a branch exists at version 1.2. Adding new deltas at two different places in the SCCS file creates programs for two environments that have some similar code (1.1 and 1.2) and some code that is not the same.

Creating another branch from delta 1.2 builds a second series of program files. The new branch begins with delta **1.2.2.1**.

To create a branch delta, use the **-r** flag and specify the release and level where the branch occurs. In the next example, deltas 1.3 and 1.4 already exist.

```
$ get -e -r1.3 s.test.c
1.3
new delta 1.3.1.1
67 lines
```

Create deltas on branches using the same methods.

Getting Duplicate File Versions

To edit a file, get the file version using the **get** command (with the **-e** flag) and save the changes with the **delta** command. Several different editable versions of an SCCS file can exist as long as each one is in a different directory. If you try to get the same editable file version more than once into the same directory without using the **delta** command, SCCS writes an error message.

To get the same editable file version more than once, set the **j** option in the SCCS file with the **admin** command. Set the **j** option using the **-f** flag. You can then get the same SID several times from different directories, creating a separate file for each **get** command. Although the files originate from a single SID, SCCS gives each of them a unique new SID.

In the following example, the **pwd** command displays the current directory. Then the **j** option is set with the **admin** command.

```
$ pwd
   /u/dan/sccs
$ admin -fj s.test.c
```

Then use the **get** command to retrieve the latest version of the file.

```
$ get -e s.test.c
1.1
new delta 1.2
5 lines
```

Change to directory `/u/new`, and issue the **get** command again.

```
$ cd /u/new
$ get -e /u/dan/sccs/s.test.c
1.1
new delta 1.1.1.1
5 lines
```

Notice that SCCS creates two deltas, 1.2 and 1.1.1.1, from the single original file version of 1.1. Look at the `p.test.c` file. It shows a separate entry for each version currently in use. The `p.test.c` file remains in the directory until you take care of both file versions with either the **delta** command or the **unget** command.

Using the delta Command

The **delta** command saves the changes made to a particular version of an SCCS file. To use the **delta** command:

1. Use **get -e** to get an editable version of the file
2. Edit that file
3. Use **delta** to create a new version of the SCCS file.

When using the **delta** command, it prompts for comments. The comments are for that particular delta and appear in the SCCS file header. The comments are not retrieved when you **get** the delta and do not appear in the text of a retrieved file. Use comments to keep track of why a delta was created.

To see the comments, use an editor to look at the SCCS file, write the SCCS file to the display screen with the **cat** command, or print selected parts of the file to standard output using the **prs** command. Refer to *AIX Operating System Commands Reference* for descriptions of these commands. Remember not to change the contents of the SCCS file directly. To change the delta comments, use the **cdc** command.

A common use of the **delta** command is shown below:

```
$ delta s.test.c
Enter comments, terminated with EOF or blank line:
```

Then enter comments, as follows:

```
This delta contains the payroll function
```

delta then finishes processing and displays:

```
1.4
24 inserted
3 deleted
45 unchanged
```

The preceding example stores the comment in the SCCS file header and creates delta 1.4. It then lists how many lines of text are inserted, deleted, or unchanged. SCCS may give unexpected numbers for these categories because of its definition for an edited line of text. However, the number of lines inserted plus the number of lines left unchanged should equal the total number of lines in the file.

SCCS does not allow using the **delta** command if an editable file does not exist. However, once an editable file exists (created with **get -e**), SCCS creates the delta without checking the data being stored in the file.

Note: When using identification keywords in SCCS files, do not use the **delta** command with a file built as read-only if an editable version of the file also exists. When you get a file as read-only, SCCS replaces identification keywords with their values. Using the **delta** command on the file saves the values and the identification keywords are lost. To recover, remove the delta, or re-edit the file and replace the identification keywords.

To prevent the loss of keywords, use the **admin** command with the **-f** flag to specify the **i** option.

Using the help Command

SCCS provides a limited form of help for certain error codes and all of the SCCS commands. To get help on a specific command or error code, use the following format:

```
help [command].. [code]..
```

The **help** program prompts for a `command` or an `error code` if those parameters are not included in the command. If it does not have information about a specific error code, **help** writes an error message and continues processing. For example, to get help on **rmdel** and two error codes, enter the following:

```
$ help rmdel gee ge5
```

The **help** command replies:

```
rmdel:
        rmdel -r<SID> <file> ...

ERROR: gee not found (hel)

ge5:
"nonexistent sid"
The specified sid does not exist in the given file.  Check for typos.
```

The response indicates that either the **help** command does not have information for the error code `gee` or the code does not exist.

Chapter 11. Finding and Changing Strings

CONTENTS

About This Chapter 11-2
Finding Strings 11-3
 Strings 11-3
 Example of Commands 11-5
Scanning Files 11-6
 Program File 11-6
 Variables 11-8
 BEGIN and END 11-8
 Using Regular Expressions as Patterns 11-9
 Using Relational Expressions as Patterns 11-11
 Using Combinations of Patterns 11-11
 Using Pattern Ranges 11-12
 Using Functions within an Action 11-12
 Using Variables in an Action 11-13
 Using Operators in an Action 11-14
 Using Field Variables in an Action 11-14
 Concatenating Strings 11-15
 Using Arrays 11-15
 Using Control Statements 11-16
Editing Files with sed 11-18
 Starting the Editor 11-18
 How sed Works 11-19
 Selecting Lines for Editing 11-20
 Regular Expressions 11-21
 sed Command Summary 11-23
 Text in Commands 11-28
 String Replacement 11-29

About This Chapter

This chapter contains introductory information to some of the system programs that are useful when developing a program on an RT work station. These programs are not required to create programs, but they do provide added services that make checking and maintaining programs easier. Use these programs in **shell** programs that you develop. The commands described in this chapter:

- Find a specified series of characters in a text file.
- Find and change information in a text file.
- Make fast editing changes on a large text file.

Complete reference information about all commands is in *AIX Operating System Commands Reference*.

Finding Strings

The system provides three similar programs to help locate a series of characters (***string***) in a file. These programs are:

grep A general function program that finds literal strings in a file, and also finds strings that you specify using wildcard characters. (These are not the same as the shell wildcard characters).

fgrep A faster version of **grep** that only finds literal strings in a file (wildcards are not allowed).

egrep An extended version of **grep** to look for more complex expressions.

Use these programs to search one or more files at a time to answer the following questions:

- In which file(s) does the string occur (**-l** flag)?
- On how many lines in each file does the string occur (**-c** flag)?
- What is the line number of each place that the string occurs (**-n** flag)?
- What lines in which file(s) contain the string (no flag)?
- What lines in which file(s) exactly match the string (**-x** flag)?
- What lines in which file(s) do not contain the string (**-v** flag)?
- What is the disk block number of each place that the string occurs (**-b** flag)?

Strings

A ***string*** is any group of characters to find. Enclose the string in ' ' (single quotes) to ensure that the shell does not interpret blanks or special shell characters in the string as part of its syntax.

For example, because blanks separate the parameters on the command line, the shell interprets the command:

```
fgrep find me myfile
```

as a request to find the string, `find`, in the files `me` and `myfile`. To find the string, `find me`, in `myfile` specify the command like:

```
fgrep 'find me' myfile
```

Literal Strings

A literal string is a string that does not contain wildcard characters, and can, therefore, be interpreted just as it is. The previous example contains a literal string, `find me`. Use literal strings to specify exactly what to find.

For example, to find the file that contains the module, `eprog()`, in five source files that contain several modules each, use the command:

```
fgrep 'eprog()' file1 file2 file3 file4 file5
```

This command finds not only the actual module but any references to it. Look at the output lines to determine which reference actually contains the module definition. To find only the module definition, use the **-x** flag:

```
fgrep -x 'eprog()' file1 file2 file3 file4 file5
```

For coding formats that put each module definition left-justified on a line by itself, this command writes out only that line, together with the name of the file in which it occurs.

Regular Expressions

A regular expression is a string that contains wildcard characters and operators that define a set of one or more possible strings. The find string programs use a set of wildcard characters that is different from the shell wildcards, but the same as the line editor, **ed**. These wildcard characters and operators are:

.(period)
Specifies any character except new-line.

^ (caret)
Specifies the beginning of the line when it is the first character in a regular expression.

$ (dollar sign)
Specifies the end of the line when it is the last character in a regular expression.

[] (square brackets)
Encloses a set of characters (not empty) that represents any **one** of the characters in the set.

[abc]	Represents either a or b or c
[a-c]	Represents all characters in the current collation sequence (as defined by the **NLFILE** or **NLCTAB** environment variable) that collate between *a* and *c*, inclusive.
[:alpha:]	Represents all characters belonging to the character class. (Refer to the **ed** program in *coms* for details about supported character classes.)

| (vertical bar)

OR - indicates a search for either one string or another:

```
egrep 'prog()|prog1()' file1
```

finds lines containing either `prog()` or `prog1()`.

Refer to the **ed** program in *AIX Operating System Commands Reference* for details for building regular expressions.

Example of Commands

To check a C program for proper nesting of braces, use the following two commands:

```
grep -c '{' file1
```

and

```
grep -c '}' file1
```

Each command responds with a number that represents the number of lines in the file containing either { or }, respectively. If the numbers are not the same, you may have a problem. Check the file again using a different form of the command:

```
egrep '{|}' file1
```

This command displays each line in the file that contains either a { or a }. It displays the lines in the order that they occur in the file, so that you can quickly check for matching pairs of open and closed braces.

Scanning Files

The **awk** program is an extension of the features of **grep**. It performs the following operations:

- Scans a file or list of files to find matches to a ***regular expression***.
- Performs an operation on the lines that are found, defined by an ***action***.

It uses all of the regular expression building features that **egrep** uses, plus it allows you to:

- Write selected fields of the line
- Calculate running totals
- Change syntax in a program source file
- Change system calls when porting from one system to another.

The **awk** program finds and changes strings in text files. In addition, it provides numeric processing, variables, more general pattern selection for finding strings, and flow control statements. This program handles both string and numeric data. In general, this program is useful for:

- Processing input to find numeric counts, sums or subtotals
- Verifying that the contents of a field contains only numeric information
- Checking to see that delimiters are balanced in a programming file
- Processing data contained in fields within lines
- Changing data from one program into a form that can be used by a different program.

Program File

When using **awk**, you can either enter the search pattern directly on the command line as with **grep**, or you can build a file that contains both the search pattern and the actions to perform. Using a program file puts many patterns in one file, and saves typing the command again to correct an error in the search pattern. When using a program file, run **awk** with the **-f** flag, such as:

```
awk -f pfile file1 file2 file3
```

In this command, `pfile` is the name of the program file, `file1`, `file2`, and `file3` are the files to be searched, and **-f** tells **awk** to look in `pfile` for the search program.

The program file is a series of statements that look like:

```
pattern          { action }
pattern          { action }
pattern          { action }
    .
    .
    .
```

Where:

pattern Is a regular expression, or series of regular expressions, that defines the search pattern, including:

- Boolean combinations of regular expressions using the operators **!** **||** **&&** and **()** .
- Boolean combinations of relational operators on strings, numbers, fields, variables, and array elements.

action Is a set of steps to perform on the line, designated with **awk** commands and operators, including:

- Any expressions that are used in a ***pattern***
- Arithmetic and string expressions
- Assignment statements
- If-else statements
- While-for statements
- More than one output stream.

{ } Are delimiters that set off the action from the search pattern.

In any line, you can omit either the pattern or the action. If you omit the pattern, **awk** performs the action on all lines in the file(s); if you omit the action, **awk** copies the line to standard output.

When **awk** runs, it reads the first line of the input data file and matches it against each of the ***patterns*** in the program file in the order that they appear in the program file. When **awk** finds a pattern that matches the line, it performs the associated ***action*** on that line. Then it continues to search for more matches in the program file. When it has compared the first input line against all patterns in the program file, **awk** reads the next input line and starts at the beginning of the program file with that line.

Variables

Awk recognizes the following built-in variables:

FILENAME	The name of the current input file.
NR	The number of the current record.
NF	The number of fields in the current record.
FS	The character used for a field separator.
RS	The character used for a record separator.
$0	The contents of the input record.
$n	The contents of field *n* of the input record.
OFS	The character used for output field separator (the character between fields when the data is written; a blank if you do not change it).
ORS	The character used for output record separator (the character between records when the data is written; a new-line if you do not change it).

BEGIN and END

The **awk** program recognizes two special keywords that define the beginning (**BEGIN**) and the end (**END**) of the input file. The pattern **BEGIN** matches the beginning of the input before reading the first record. Therefore, **awk** performs any actions associated with this pattern once, before processing any of the input file. **BEGIN** must be the first pattern in the program file. For example, to change the field separator to a colon for all records in the file, include the following line as the first line of the program file:

```
BEGIN   {FS=":"}
```

Similarly, the pattern, **END**, matches the end of the input file after processing the last record. Therefore, **awk** performs any actions associated with this pattern once, after processing all of the input file. **END** must be the last pattern in the program file. For example, to print the total number of lines in the input file, include the following line as the last line in the program file:

```
END     {print NR}
```

Using Regular Expressions as Patterns

The simplest regular expression is a literal string of characters, enclosed in slashes. For example, if the program file contains only the entry:

```
/the/
```

the file is a complete program that displays all lines containing the string `the`. Because the string does not specify any blanks or other qualifiers, the program also displays lines containing words such as:

```
theater
northern
```

that have the string as part of them. The program is sensitive to uppercase and lowercase, and only displays lines containing the lowercase form of the string.

Square Bracket Expression

To find lines that contain `The` (the string as the first word in a sentence) in addition to the lowercase version, use a square bracket expression to represent either uppercase or lowercase. A ***square bracket expression*** is a set of characters, enclosed in `[ ]` (square brackets). Each character in the square bracket expression satisfies the search conditions for *one* character position. For example, to find lines containing both forms of the word `the` (and words containing it), use the string:

```
/[Tt]he/
```

The following string:

```
/[cCdDhH]ome/
```

finds lines that contain the words:

```
come
Come
dome
Dome
home
Home
```

Use ranges within a bracket expression to indicate a range of characters. To define a range, enter the following:

1. `[`
2. The first character of the range
3. `-` (minus)

4. The last character of the range
5.]

Ranges specify a sequence of continuous characters as defined in the current collation table (see "Overview of International Character Support" in *IBM RT Managing the AIX Operating System*).

You can also use character types, such as `[:alpha:]`. The **awk** program supports the same character class expressions as **grep** and **egrep**.

Special Characters

The **awk** program defines the symbols shown in Figure 11-1 to use in building patterns:

Symbol	Meaning
/	String delimiter: Indicates the start and end of a string or regular expression.
\	Escape: Tells **awk** to treat the next character as a regular ASCII character instead of a symbol that **awk** treats as a special character.
$0	Matches the entire line with the pattern.
$n	Matches field *n* (*n* is an integer) in each input line.
~	Match field operator: Tells **awk** to match the regular expression with a specified field in each line, not the line.
!~	Not match field operator: Tells **awk** to compare the regular expression with a specified field in each line and perform the action only if the expression does not match the field.
^	Beginning of the line or field: When placed at the start of a string, tells **awk** to match the string only when it occurs at the start of a line or specified field.
$	End of the line or field: When placed at the end of a string, tells **awk** to match the string only when it occurs at the end of a line or specified field.

Figure 11-1. awk Special Characters

Using Relational Expressions as Patterns

Use a relational expression as a pattern in the program file. The **awk** program defines the following relational operators for use in building patterns:

<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	Equivalent
!=	Not equivalent

Examples of Relational Expressions in a Pattern

To find all lines where the second field ($2) is at least 100 greater than the first field:

```
$2 > $1 + 100
```

To find lines that contain an even number of fields:

```
NF % 2 == 0
```

To find lines that begin with `s, t, u,...`:

```
$1 >= "s"
```

To perform a string comparison between the first two fields:

```
$1 > $2
```

Using Combinations of Patterns

Combine two or more patterns using Boolean operators:

\|\|	Or
&&	And
!	Not

For example, the pattern:

```
$1 >= "T" && $1 < "U" && $1 != "The"
```

finds lines that begin with `T`, and are not the word `The`.

Using Pattern Ranges

A pattern range allows the use of one pattern to begin an action on the lines of text, and another pattern to end the action on lines of text. Specify a pattern range by using two patterns separated by commas. The first pattern specifies the starting pattern; the second pattern specifies the ending pattern. Therefore, the line:

```
/The/,/End/  {action}
```

finds the first line that contains the pattern `The` and begins performing the `action` on all lines following it in the file until **awk** finds a line containing the pattern `End`. **Awk** does not change either the line containing `The` or the line containing `End`.

Similarly, the line:

```
NR==100,NR==200        {action}
```

performs the action starting at line 100 and ending at line 200 of the input file.

Using Functions within an Action

The **awk** program provides the following functions to use within an action:

length	Returns the length in characters of the current record.
length(arg)	Returns the length in characters of the string specified by **arg**.
blength	Returns the length in bytes of the current record.
blength(arg)	Returns the length in bytes if the string specified by **arg**.
sqrt(arg)	Returns the square root of **arg**
log(arg)	Returns the base e logarithm (natural logarithm) of **arg**.
exp(arg)	Returns the exponential part of **arg**.
int(arg)	Returns the integer part of **arg**.
substr(s,m) [**n**]	Returns a string that is part of string **s**, beginning at character **m** and continuing for **n** characters (or the end of string **s**). If **m** is 1, the string starts at the beginning of string **s**. If you do not supply a value for **n**, the string continues to the end of string **s**.
index(s1,s2)	Returns the character position in string **s1** where string **s2** occurs. If **s2** is not in **s1**, this function returns a zero.
sprintf(f,e1,e2,...)	Formats the argument strings (**e1, e2, ...**) using the format specification **f**, and returns a formatted string. The parameters are:

f
A formatting specification string defined using the formatting specifications of the **printf** library routine.

e1,e2,...
A series of strings that the **f** format specification acts upon.

printf(f,e1,e2,...) Prints the formatted string **f**, on **stdout**

print(arg) Prints **arg** on **stdout**

split(s,array) [sep] Splits the string *s* into consecutive elements of the array[1] . . . [n] and returns the number of elements. If you provide the **sep** argument, it is either the field separator or the character defined by the **FS** variable.

Using Variables in an Action

The **awk** program sets all variables in actions to zero when it begins executing the action. The variables do not have a strict type; they take on numeric (floating point) values or string values depending on their use in the action expression. For example, the expression:

```
x = 1
```

indicates that x is a numeric variable. Similarly, the expression:

```
x = "smith"
```

indicates that x is a string variable. However, **awk** converts between strings and numbers when needed. Therefore, the expression:

```
x = "3" + "4"
```

assigns a value of 7 (numeric) to x, even though the arguments are literal strings. If **awk** cannot change a string variable to numeric when you are using it as a numeric variable, **awk** assigns it a numeric value of zero. To force a variable to be treated as a single type:

string Add the null string (" ") to the value assigned to the variable

numeric Add zero (0) to the value assigned to the variable.

Using Operators in an Action

Use the following operators to build expressions within the action statement:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remaindering)
+ +	Increment
--	Decrement
+ =	Increment by value
- =	Decrement by value
* =	Multiply by value
/ =	Divide by value
% =	Modulo by value
~	Match string
!~	Not match string

For example, to find and print the sum of all the first fields and the sum of all the second fields in a file with the program file:

```
                {s1 += $1; s2 += $2}
END             {print s1,s2}
```

Using Field Variables in an Action

Fields in **awk** share the same properties as variables. They can be used in arithmetic or string operations and can be assigned to a numeric or string value. For example, to replace the first field with a sequence number:

```
{$1 = NR;print}
```

To accumulate two fields into a third field:

```
{$1 = $2 + $3;print $0}
```

Use numeric expressions for field references, such as:

```
{print $i,$(i+1),$(i+n)}
```

How you use a field determines whether **awk** treats a field as numeric or string. If it cannot tell how the field is used, **awk** treats fields as strings.

awk splits input lines into fields as needed. You can also split any variable or string into fields. For example:

```
n = split(s,array,sep)
```

splits the string `s` into `array[1]...array[n]` and returns the number of elements. If you provide the `sep` argument, it is the field separator. If you do not provide `sep`, the field separator is the character defined by the variable **FS**.

Concatenating Strings

Concatenate strings by placing their variable names together in an expression. For example, the expression:

```
length($1 $2 $3)
```

Returns the length in characters of the first three fields. The expression:

```
print $1 " is " $2
```

Prints the first two fields separated by `" is "`. You can use variables and numeric expressions when concatenating strings.

Use **blength** to get the length in bytes.

Using Arrays

You do not need to declare array elements. **awk** sets them to zero when first used. Use any value that is not null, including a string value, for a subscript. An example of the numeric subscript is:

```
x[NR] = $0
```

This expression assigns the current input record to the NRth element of the array `x`. For an example of using a string subscript, suppose that the input contains fields with values like `apple` or `orange`. Then the program:

```
/apple/         {x["apple"]++}
/orange/        {x["orange"]++}
END             {print x["apple"], x["orange"]}
```

increments counts for the named array elements and prints them at the end of the input.

Problems can occur when you use an **if** or **while** statement to locate an array element. If the array subscript does not exist, the following statement:

```
{if (exists[$2] == 1 )}
```

adds the subscript as a new b tree node with a null value. To avoid this type of problem use the fragment below when *i* is printed with a **for** loop after the **if** or **while** is used with a relational operator.

```
for (i in exists){
        if (exists[i] != ""){
                print i;
        }
}
```

Using Control Statements

The **awk** language also provides the following control structures as in the C language:

- If-else
- While
- For
- Break
- Continue
- Next
- Exit
- Braces for statement grouping
- Comments.

If-Else Statement

The **if-else** statement is exactly like that of the C language. The condition in parentheses of an **if-else** statement is evaluated; if it is true, the statement following the **if** is performed. The **else** part is optional.

While Statement

The **while** statement is exactly like that of the C language. For example, to print all input fields, one on each line, use the following program:

```
i = 1
while(i<=NF)
{
        print $i
        ++i
}
```

For Statement

The **for** statement is also like that of the C language. For example, the previous **while** example could also be written:

```
for(i=1;i<=NF;++i)
        print $i
```

Break Statement

The **break** statement causes an immediate exit from an enclosing **while** or **for** loop.

Continue Statement

The **continue** statement causes the next iteration of an enclosing loop to begin.

Next Statement

The **next** statement causes **awk** to skip to the next input record and begin scanning the patterns from the top of the program file.

Exit Statement

The **exit** statement causes the program to stop as if the end of the input occurred.

Comments

Include comments in the **awk** program file to explain program logic. Comments begin with the # character and end with the end of the line. For example:

```
print x,y       #this is a comment
```

Editing Files with sed

The **sed** program is a text editor that has similar functions to those of **ed**, the line editor. Unlike **ed**, however, the **sed** program performs its editing without interacting with the person requesting the editing. This method of operation allows **sed** to:

- Edit very large files
- Perform complex editing operations many times without requiring extensive retyping and cursor positioning (as interactive editors do)
- Perform global changes in one pass through the input.

The editor keeps only a few lines of the file being edited in memory at one time, and does not use temporary files. Therefore, the file to be edited can be any size as long as there is room for both the input file and the output file in the file system.

Starting the Editor

To use the editor, create a command file containing the editing commands to perform on the input file. The editing commands perform complex operations and require a small amount of typing in the command file. Each command in the command file must be on a separate line. Once the command file is created, enter the following command on the command line:

```
sed -fcmdfile >output <input
```

In this command the parameters mean:

cmdfile The name of the file containing editing commands.

output The name of the file to contain the edited output.

input The name of the file, or files, to be edited.

The **sed** program then makes the changes and writes the changed information to the output file. The contents of the input file are not changed.

How sed Works

The **sed** program is a stream editor that receives its input from standard input, changes that input as directed by commands in a command file, and writes the resulting stream to standard output. If you do not provide a command file and do not use any flags with the **sed** command, the **sed** program copies standard input to standard output without change. Input to the program comes from two sources:

Input stream A stream of ASCII characters either from one or more files or entered directly from the keyboard. This stream is the data to be edited.

Commands A set of addresses and associated commands to be performed, in the general form of:

```
[line1 [,line2] ] command [argument]
```

The parameters `line1` and `line2` are called ***addresses***. Addresses can be either patterns to match in the input stream, or line numbers in the input stream as explained in "Selecting Lines for Editing" on page 11-20.

You can also enter editing commands along with the **sed** command by using the **-e** flag.

When **sed** edits, it reads the input stream one line at a time into an area in memory called the ***pattern space*** as shown in Figure 11-3 on page 11-20. When a line of data is in the pattern space, **sed** reads the command file and tries to match the addresses in the command file with characters in the pattern space. If it finds an address that matches something in the pattern space, **sed** then performs the command associated with that address on the part of the pattern space that matched the address. The result of that command changes the contents of the pattern space, and thus becomes the input for all following commands.

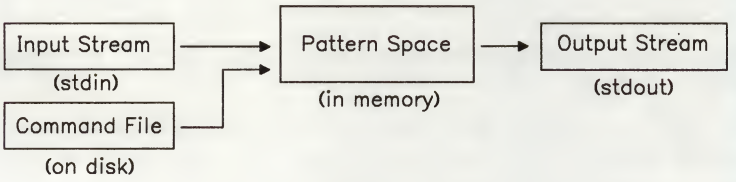
When **sed** has tried to match all addresses in the command file with the contents of the pattern space, it writes the final contents of the pattern space to standard output. Then it reads a new input line from standard input, and starts the process over at the start of the command file.

Some editing commands change the way the process operates. See the **Control** commands in Figure 11-6 on page 11-24.

Flags used with the **sed** command can also change the operation of the command as shown in Figure 11-2 on page 11-20.

Flag	Function
-n	**sed** does not copy all input lines to standard output. Instead, it copies only those lines that editing commands specifically write to standard output. See the **print lines** and **substitution** commands in Figure 11-6 on page 11-24.
-e	**sed** uses the argument that directly follows this flag as an editing command.
-f	**sed** uses the argument that directly follows this flag as the name of the file containing the editing commands. This file must contain editing commands with each command on a separate line.

Figure 11-2. sed Command Flags



A5AC6032

Figure 11-3. sed Block Diagram

Selecting Lines for Editing

Use one of the following forms of addressing to select lines in the input stream for editing:

Line numbers As the editor reads each input line, it increments its line counter starting with line 1 as the first line of the first file in the input stream. The line counter runs cumulatively through all files in the input stream. The editor does not reset the counter when it opens a new file in the same input stream. The value of the line counter for each line is the ***line number*** for that line.

Specifying a decimal integer as either `line1` or `line2` in the editing commands indicates the line number of the line to be edited. The character **$** matches the last line of the last file in the input stream.

Context addresses A context address is a ***regular expression*** enclosed in / (slashes). See "Regular Expressions" on page 11-21 for a description of the regular expressions that **sed** recognizes. The whole regular expression must match some part of the pattern space for a successful context address match.

Editing commands can have zero, one, or two addresses, depending on the command and how you use it. The number of addresses determines how the address is used:

Addresses	Use of Command
No address	The command is applied to every line in the input stream.
One address	The command is applied to each line that matches the address.
Two addresses	The command is applied to the range of addresses starting with the line that matches the first address up to and including the first line that matches the second address. The editor then tries to match the first address again to find another range. Separate two addresses with a comma as shown in the syntax diagrams in Figure 11-6 on page 11-24.

Regular Expressions

A regular expression is a string that contains literal characters, wildcard characters and/or operators that define a set of one or more possible strings. The stream editor uses a set of wildcard characters that is different from the shell wildcards, but the same as the line editor, **ed**. These wildcard characters and operators are shown in Figure 11-4 on page 11-22.

Symbol	Function
.	A period specifies any character except new-line.
^	A caret specifies the beginning of the line when it is the first character in a regular expression.
$	A dollar sign specifies the end of the line when it is the last character in a regular expression.
[]	Square brackets enclose a set of characters (not empty) that represents any **one** of the characters in the set. If the first character inside the brackets is a ^ (caret), the regular expression matches any character *except* the characters in the set and the new-line at the end of the pattern space. **[abc]** Represents either a or b or c **[a-c]** Represents all characters in the current collation sequence between *a* and *c*, inclusive. **[A-z]** Is the same as [abc] (not **fgrep**). The hyphen defines a range of ASCII vales that starts with the value of the first letter and ends with the value of the second letter. **[[:alpha:]]** Represents all characters in the character class alpha (see **ctype** in *AIX Operating System Technical Reference*).
\n	A new-line character matches a new-line character that is not the new-line character at the end of the pattern space.
*	A regular expression followed by an asterisk matches any number (including 0) of adjacent occurrences of that regular expression.
\(and \)	A set of backslash-parentheses enclose a regular expression that can be repeated using the \d expression.
\d	**d** is a single ASCII digit. This symbol in *string* is replaced by the set of characters in the input lines that matches the dth substring in *pattern*. Substrings begin with the characters \(and end with the characters \). See "String Replacement" on page 11-29 for more information about using this expression.
//	The null string is the same as the last regular expression in the edit stream.

Figure 11-4 (Part 1 of 2). sed Wildcard Characters

Symbol	Function
\	A backslash tells **sed** to treat the next character as a regular ASCII character instead of a symbol that **sed** treats as a special character.

Figure 11-4 (Part 2 of 2). sed Wildcard Characters

Refer to the **ed** program in *AIX Operating System Commands Reference* for details for building regular expressions.

sed Command Summary

All **sed** commands are single letters plus some parameters, such as line numbers or text strings. Figure 11-6 on page 11-24 summarizes the commands that make changes to the lines in the pattern space. The table uses the symbols shown in Figure 11-5 in the syntax diagrams:

Symbol	Meaning
[]	Square brackets enclose optional parts of the commands
italics	Parameters in italics represent general names for a name that you enter. For example, *filename* represents a parameter that you replace with the name of an actual file.
line1	This symbol is a line number or regular expression to match that defines the starting point for applying the editing command.
line2	This symbol is a line number or regular expression to match that defines the ending point to stop applying the editing command.

Figure 11-5. Syntax Symbols

Category	Function	Syntax/Description
Line Manipulation:	Append lines	[*line1*]a\\n*text* Writes the lines contained in *text* to the output stream after *line1*. The **a** command must appear at the end of a line. See "Text in Commands" on page 11-28 for the format of the *text*.
	Change lines	[*line1* [,*line2*]]c\\n*text* Deletes the lines specified by *line1* and *line2* as the **delete lines** command does. Then it writes *text* to the output stream in place of the deleted lines.
	Delete lines	[*line1* [,*line2*]]d Removes lines from the input stream and does not copy them to the output stream. The lines not copied begin at line number *line1*. The next line copied to the output stream is line number *line2* + *1*. If you specify only one line number, then only that line is not copied. If you do not specify a line number, the next line is not copied. You cannot perform any other functions on lines that are not copied to the output.
	Insert lines	[*line1*]i\\n*text* Writes the lines contained in *text* to the output stream before *line1*. The **i** command must appear at the end of a line. See "Text in Commands" on page 11-28 for the format of the *text*.
	Next line	[*line1* [,*line2*]]n Reads the next line, or group of lines from *line1* to *line2* into the pattern space. The current contents of the pattern space are written to the output if it has not been deleted.

Figure 11-6 (Part 1 of 5). sed Command Summary

Category	Function	Syntax/Description
Substitution:	Substitute for Pattern	[*line1* [,*line2*]]s/*pattern*/*string*/*flags* Searches the indicated line(s) for a set of characters that matches the regular expression defined in *pattern*. When it finds a match, the command replaces that set of characters with the set of characters specified by *string*. See "String Replacement" on page 11-29 for specifications for this command.
Input and Output:	Print lines	[*line1* [,*line2*]]p Writes the indicated lines to **stdout** at the point in the editing process that the **p** command occurs.
	Write lines	[*line1* [,*line2*]]w *filename* Writes the indicated lines to *filename* at the point in the editing process that the **w** command occurs. If *filename* exists, it is overwritten; otherwise, it is created. A maximum of 10 different files can be mentioned as input or output files in the entire editing process. Include exactly one space between **w** and *filename*.
	Read file	[*line1*]r *filename* Reads *filename* and appends the contents after the line indicated by *line1*. Include exactly one space between **r** and *filename*. If *filename* cannot be opened, the command reads it as a null file without giving any indication of an error.
Matching Across Lines:	Join next line	[*line1* [,*line2*]]N Joins the indicated input lines together, separating them by an imbedded new-line character. Pattern matches can extend across the imbedded new-line(s).

Figure 11-6 (Part 2 of 5). sed Command Summary

Category	Function	Syntax/Description
	Delete first line of pattern space	[*line1* [,*line2*]]D Deletes all text in the pattern space up to and including the first new-line character. If only one line is in the pattern space, reads another line. Starts the list of editing commands again from the beginning.
	Print first line of pattern space	[*line1* [,*line2*]]P Prints all text in the pattern space up to and including the first new-line character to **stdout**.
Pick up and Put down:	Pick up copy	[*line1* [,*line2*]]h Copies the contents of the pattern space indicated by *line1* and *line2* if present, to the holding area. The previous contents of the holding area are destroyed.
	Pick up copy, appended	[*line1* [,*line2*]]H Copies the contents of the pattern space indicated by *line1* and *line2* if present, to the holding area, and appends it to the end of the previous contents of the holding area.
	Put down copy	[*line1* [,*line2*]]g Copies the contents of the holding area to the pattern space indicated by *line1* and *line2* if present. The previous contents of the pattern space are destroyed.
	Put down copy, appended	[*line1* [,*line2*]]G Copies the contents of the holding area to the end of the pattern space indicated by *line1* and *line2* if present. The previous contents of the pattern space are not changed. A new-line character separates the previous contents from the appended text.

Figure 11-6 (Part 3 of 5). sed Command Summary

Category	Function	Syntax/Description
	Exchange copies	[*line1* [,*line2*]]x Exchanges the contents of the holding area with the contents of the pattern space indicated by *line1* and *line2* if present.
Control:	Negation	[*line1* [,*line2*]]! The ! (exclamation point) applies the command that follows it on the same line to the parts of the input file that are *not* selected by *line1* and *line2*.
	Command groups	[*line1* [,*line2*]]{ *grouped commands* } The { (left brace) and the } (right brace) enclose a set of commands to be applied as a set to the input lines selected by *line1* and *line2*. The first command in the set can be on the same line or on the line following the left brace. The right brace must be on a line by itself. You can nest groups within groups.
	Labels	:*label* Marks a place in the stream of editing commands to be used as a destination of a branch. The symbol *label* is a string of up to 8 bytes. Each *label* in the editing stream must be different from any other *label*. For a related discussion see the section on **sed** in *AIX Operating System Commands Reference*, in particular the description of the **t** subcommand.
	Branch to label, unconditional	[*line1* [,*line2*]]b*label* Branches to the point in the editing stream indicated by *label* (see **:label** above) and continues processing the current input line with the commands following *label*. If *label* is null, branches to the end of the editing stream, which results in reading a new input line and starting the editing stream over. The string *label* must appear as a label in the editing stream.

Figure 11-6 (Part 4 of 5). sed Command Summary

Category	Function	Syntax/Description
	Test and Branch	[*line1* [,*line2*]]t*label* If any successful substitutions were made on the current input line, branches to *label*. If no substitutions were made, does nothing. Clears the flag that indicates a substitution was made. This flag is cleared at the start of each new input line.
	Quit	[*line1*]q Stops editing in an orderly fashion by: • Writing the current line to the output • Writing any appended or read text to the output • Stopping the editor.
	Find line number	[*line1*]= Writes to standard output the line number of the line that matches *line1*.

Figure 11-6 (Part 5 of 5). sed Command Summary

Text in Commands

The **append**, **insert** and **change** lines commands all use a supplied text string to add to the output stream. This text string conforms to the following rules:

- Can be one or more lines long.
- Each \n (new-line character) inside *text* must have an additional \ character before it (\\n).
- The *text* string ends with a new-line that does not have an additional \ character before it (\n).
- Once the command inserts the *text* string, the string:
 - Is always written to the output stream, regardless of what other commands do to the line that caused it to be inserted.
 - Is not scanned for address matches.
 - Is not affected by other editing commands.
 - Does not affect the line number counter.

String Replacement

The **s** command performs string replacement in the indicated lines in the input file. If the command finds a set of characters in the input file that satisfies the regular expression *pattern*, it replaces the set of characters with the set of characters specified in *string*.

The *string* parameter is a literal set of characters (digits, letters and symbols). Two special symbols can be used in *string*:

& This symbol in *string* is replaced by the set of characters in the input lines that matched *pattern*. For example, the command:

```
s/boy/&s/
```

tells **sed** to find a pattern `boy` in the input line, and copy that pattern to the output with an appended `s`. Therefore, it changes the input line:

From: `The boy look at the game.`
To: `The boys look at the game.`

\d **d** is a single digit. This symbol in *string* is replaced by the set of characters in the input lines that matches the `d`th substring in *pattern*. Substrings begin with the characters `\(` and end with the characters `\)`. For example, the command:

```
s/\(stu\)\(dy\)/\1r\2/
```

tells **sed** to find a pattern `study` in the input line, and copy that pattern to the output with an `r` inserted in the middle. Therefore, it changes the input line:

From: `The study chair`
To: `The sturdy chair`

The letters that appear as flags change the replacement as follows,

g Substitute *string* for all instances of *pattern* in the indicated line(s). Characters in *string* are not scanned for a match of *pattern* after they are inserted. For example, the command:

```
s/r/R/
```

changes:

From: `the round rock`
To: `the Round rock`

But, the command:

```
s/r/R/g
```

changes:

From: `the round rock`
To: `the Round Rock`

p Print (to **stdout**) the line that contains a successfully matched *pattern*.

w *filename* Write to *filename* the line that contains a successfully matched *pattern*. If *filename* exists, it is overwritten; otherwise, it is created. A maximum of 10 different files can be mentioned as input or output files in the entire editing process. Include exactly one space between **w** and *filename*.

Chapter 12. Using the Macro Processor (m4)

CONTENTS

About This Chapter 12-2
Using the Macro Preprocessor 12-3
Defining Macros 12-4
 Using the Quote Characters 12-5
 Arguments 12-6
Using Other m4 Macros 12-8
 Changing the Quote Characters 12-10
 Removing a Macro Definition 12-11
 Checking for a Defined Macro 12-11
 Using Integer Arithmetic 12-12
 Manipulating Files 12-13
 Redirecting Output 12-13
 Using System Programs in a Program 12-14
 Using Unique File Names 12-14
 Using Conditional Expressions 12-15
 Manipulating Strings 12-16
 Printing 12-17

About This Chapter

The **m4** macro processor is a front-end processor for a compiled (or assembled) programming language. The **#define** statement in C language is an example of the facility provided by the macro processor.

At the beginning of a program, you can define a symbolic name or symbolic constant as a particular string of characters. The **m4** macro processor then replaces later unquoted occurrences of the symbolic name with the corresponding string. Besides replacing one string of text with another, the **m4** macro processor provides the following features:

- Arithmetic capabilities
- File manipulation
- Conditional macro expansion
- String and substring functions.

A ***token*** is a string of letters and digits. The **m4** program reads each alphanumeric token and determines if the token is the name of a macro. It then replaces the name of the macro with its defining text, and pushes the resulting string back onto the input to be rescanned. You can call macros with arguments, in which case the arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

The **m4** program provides built-in macros; you can also define new macros. Built-in and user-defined macros work the same way except that some of the built-in macros change the state of the process. Refer to "Using Other m4 Macros" on page 12-8 for a list of the macros.

Using the Macro Preprocessor

To use the **m4** macro processor, enter the following command:

```
m4 [file]
```

The **m4** program processes each argument in order. If there are no arguments or if an argument is **−**, **m4** reads standard input as its input file. The **m4** program writes its results to standard output. Therefore, to redirect the output to a file for later use, use a command like:

```
m4 [file] >outputfile
```

Defining Macros

The **define** macro is a built-in function that defines macros. For example, if the following statement is in a program:

```
define(name, stuff)
```

the **m4** program defines the string `name` as `stuff`. When the string `name` occurs in a program file, **m4** replaces it with the string `stuff`. The string `name` must be ASCII alphanumeric and must begin with a letter or underscore. The string `stuff` is any text, but if the text contains parentheses the number of open, or left, parentheses must equal the number of close, or right, parentheses. Use the / (slash) character to spread the text for `stuff` over multiple lines. The open parenthesis must immediately follow the word **define**. For example:

```
define(N, 100)
  . . .
if (i > N)
```

defines `N` to be 100 and uses the symbolic constant `N` in a later **if** statement. Macro calls in a program have the following form:

```
name(arg1,arg2, . . . argn)
```

A macro name is only recognized if it is surrounded by nonalphanumerics. Using the following example:

```
define(N, 100)
  . . .
if (NNN > 100)
```

the variable `NNN` is not related to the defined macro `N`.

You can define macros in terms of other names. For example:

```
define(N, 100)
define(M, N)
```

defines both `M` and `N` to be 100. If you later change the definition of `N` and assign it a new value, `M` retains the value of 100, not `N`.

The **m4** macro processor expands macro names into their defining text as soon as possible. The string N is replaced by 100. Then the string M is also replaced by 100. The overall result is the same as using the following input in the first place: .

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now M is defined to be the string N, so when the value of M is requested later, the result is the value of N at that time (because the M is replaced by N, which is replaced by 100).

Using the Quote Characters

To delay the expansion of the arguments of **define**, enclose them in the ***quote*** characters. If you do not change them, the ***quote*** characters are left and right single quotes (‘ ’). See “Changing the Quote Characters” on page 12-10 to change these characters. Any text surrounded by the quote characters is not expanded immediately, but the quote characters are removed. The value of a quoted string is the string with the quote characters removed. If the input is:

```
define(N, 100)
define(M, ‘N’)
```

The quote characters around the N are removed as the argument is being collected. The result of using quote characters is to define M as the string N, not 100. The general rule is that **m4** always strips off one level of quote characters whenever it evaluates something. This is true even outside of macros. To make the word `define` appear in the output, enter the word in quote characters, as follows:

```
‘define’ = 1;
```

Another example of using quote characters is redefining N. To redefine N, delay the evaluation by putting N in quote characters. For example:

```
define(N, 100)
  . . .
define(‘N’, 200)
```

To prevent problems from occurring, quote the first argument of a macro. For example, the following fragment does not redefine N:

```
define(N, 100)
  . . .
define(N, 200)
```

The N in the second definition is replaced by 100. The result is the same as the following statement:

```
define(100, 200)
```

The **m4** program ignores this statement because it can only define names, not numbers.

Arguments

The simplest form of macro processing is replacing one string by another (fixed) string. However, macros can also have arguments, so that you can use the macro in different places with different results. To indicate where an argument is to be used within the replacement text for a macro (the second argument of its definition), use the symbol **$***n* to indicate the *n*th argument. When the macro is used, **m4** replaces the symbol with the value of the indicated argument. For example, the symbol:

```
$2
```

refers to the second argument of a macro. Therefore, if you define a macro called bump as:

```
define(bump, $1 = $1 + 1)
```

m4 generates code to increment the first argument by 1. The `bump(x)` statement is equivalent to `x = x + 1`.

A macro can have as many arguments as needed. However, you can access only nine arguments using the **$***n* symbol ($1 through $9). To access arguments past the ninth argument, use the **shift** macro which drops the first argument and reassigns the remaining arguments to the **$***n* symbols (second argument to $1, third argument to $2 ... tenth argument to $9). Using the **shift** macro more than once allows access to all arguments used with the macro.

The macro name **$0** returns the name of the macro. Arguments that are not supplied are replaced by null strings, so that you can define a macro that concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus:

```
cat(x, y, z)
```

is the same as:

```
xyz
```

Arguments $4 through $9 in this example are null since corresponding arguments were not provided.

The **m4** program discards leading unquoted blanks, tabs, or new lines in arguments, but keeps all other white space. Thus:

```
define(a, b c)
```

defines `a` to be `b c`.

Arguments are separated by commas. Use parentheses to enclose arguments containing commas, so that the comma does not end the argument. For example:

```
define(a, (b,c))
```

has only two arguments. The first argument is `a`. The second is `(b,c)`. To use a comma or single parenthesis, enclose it in quote characters.

Using Other m4 Macros

The **m4** program provides a set of macros that are already defined. The **define** macro already mentioned is one of them. Figure 12-1 lists each of these macros and provides a brief explanation of its function. The following paragraphs further explain many of the macros and how to use them.

Macro	Function
changecom(*l* , *r*)	Changes the left and right comment characters to the characters represented by *l* and *r*.
changequote(*l* , *r*)	Changes the left and right quote characters to the characters represented by *l* and *r*.
decr(*number*)	Returns the value of *number* - 1.
define(*macroname* , *replacement*)	Defines new macro *macroname* with a value of *replacement*.
defn(*macroname*)	Returns the quoted definition of *macroname*.
divert(*number*)	Changes output stream to the temporary file *number*.
divnum	Returns the value of the currently active temporaryfile.
dnl	Delete characters up to and including new-line.
dumpdef('*macroname*'...)	Prints the *macroname* and current definition of named macros.
errprint(*string*)	Prints *string* to the diagnostic output file.
eval(*expression*)	Evaluates *expression* as a 32-bit arithmetic expression.
ifdef('*macroname*' , *arg1* , *arg2*)	If macro *macroname* is defined, returns *arg1*; otherwise, it returns *arg2*.

Figure 12-1 (Part 1 of 3). m4 Built-in Macros

Macro	Function
ifelse(*string1* , *string2* , *arg1* , *arg2*)	If *string1* matches *string2*, returns the value of *arg1*; otherwise, returns the value of *arg2*.
include(*file*)	Returns the contents of the file *file*.
incr(*number*)	Returns the value of *number* + 1.
index(*string1* , *string2*)	Returns the character position in *string1* where *string2* starts (starting with character number 0), or -1 if *string1* does not contain *string2*.
len(*string*)	Returns the number of characters in *string*.
m4exit(*code*)	Exits **m4** with a return code of *code*.
m4wrap(*macroname*)	Runs macro *macroname* at the end of **m4**.
maketemp(*string...XXXXX...string*)	Creates a unique file name by replacing the characters *XXXXX* in the argument string with the current process ID.
popdef(*macroname*)	Replaces the current definition of *macroname* with the previous definition, saved with the **pushdef** macro.
pushdef(*macroname* , *replacement*)	Saves the current definition of *macroname* and then defines *macroname* to be *replacement*.
shift(*parameter list*)	Returns all but the first element of *parameter list* to perform a destructive left shift of the list.
sinclude(*file*)	Returns the contents of the file *file*, but does not report an error if it cannot access *file*.
substr(*string* , *position* , *length*)	Returns a substring of *string* that begins at character number *position* and is *length* characters long.

Figure 12-1 (Part 2 of 3). m4 Built-in Macros

Macro	Function
syscmd(*command*)	Executes the system command *command* with no return value.
sysval	Gets the return code from the last use of the **syscmd** macro.
traceoff(*macro list*)	Turns off trace for any macro in *macro list*. If *macro list* is null, turns off all tracing.
traceon(*macroname*)	Turns on trace for macro *macroname*. If *macroname* is null, turns trace on for all macros.
translit(*string* , *set1* , *set2*)	Searches *string* for characters that are in *set1*. If it finds any, changes those characters to corresponding characters in *set2*.
undefine('*macroname*')	Removes the definition of *macroname*.
undivert(*number* , *number*...)	Appends the contents of the indicated temporary files to the current temporary file.

Figure 12-1 (Part 3 of 3). m4 Built-in Macros

Changing the Quote Characters

Quote characters are normally left and right single quotes (‘ ’). If those characters are not convenient, change the quote characters with the following built-in macro:

```
changequote([,])
```

The built-in **changequote** makes the left and right brackets the new quote characters. To restore the original quote characters, use **changequote** without arguments as follows:

```
changequote
```

Removing a Macro Definition

The **undefine** macro removes the definition of some macro or built-in. For example:

```
undefine('N')
```

The macro removes the definition of N. **undefine** can also remove built-ins, as follows:

```
undefine('define')
```

Once you remove a built-in macro, you cannot use the definition of the built-in again. Single quotes are required to prevent subsitution.

Checking for a Defined Macro

The built-in **ifdef** determines if a macro is currently defined. The **ifdef** macro permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument. If the first argument is not defined, the value of **ifdef** is the third argument. If there is no third argument, the value of **ifdef** is null.

Using Integer Arithmetic

The **m4** program provides the following built-in functions for doing arithmetic on integers only:

incr Increments its numeric argument by 1

decr Decrements its numeric argument by 1

eval Evaluates an arithmetic expression

Thus, to define a variable as one more than N, use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

which defines N1 as one more than the current value of N.

The **eval** function can evaluate expressions containing the following operators (listed in decreasing order of precedence):

unary + and −

** or ^ (exponentiation)

* / % (modulus)

+ −

= = != < < = > > =

!(not)

& or && (logical and)

| or || (logical or).

Use parentheses to group operations where needed. All operands of an expression must be numeric. The numeric value of a true relation (like 1 > 0) is 1, and false is 0. The precision in **eval** is 32 bits.

For example, define M to be 2==N+1 using eval as follows:

```
define(N, 3)
define(M, 'eval(2==N+1)')
```

Use quote characters around the text that defines a macro unless the text is very simple.

Manipulating Files

To merge a new file in the input, use the built-in function: **include**. For example:

```
include(filename)
```

This function inserts the contents of *filename* in place of the include command.

A fatal error occurs if the file named in **include** cannot be accessed. To avoid a fatal error, use the alternate form **sinclude**. The built-in **sinclude** (silent include) does not write a message, but continues if the file named cannot be accessed.

Redirecting Output

The output of **m4** can be redirected to temporary files during processing, and the collected material can be output upon command. The **m4** program maintains nine possible temporary files, numbered 1 through 9. If you use the built-in macro:

```
divert(n)
```

The **m4** program writes all output from the program after the **divert** function at the end of temporary file, `n`. To return the output to the display screen, use either the **divert** or **divert(0)** command, which resumes the normal output process.

The **m4** program writes all redirected output to the temporary files in numerical order at the end of processing. The **m4** program discards the output if you redirect the output to a temporary file other than 0 through 9.

To bring back the data from all temporary files in numerical order, use the built-in **undivert**. To bring back selected temporary files in a specified order, use the built-in **undivert** with arguments. When using **undivert**, **m4** discards the temporary files that are recovered and does not search the recovered data for macros.

The value of undivert is *not* the diverted text.

The built-in **divnum** returns the number of the currently active temporary files. If you do not change the output file with the **divert** macro, **m4** puts all output in temporary file 0.

Using System Programs in a Program

You can run any program in the operating system from a program by using the **syscmd** built-in. For example, the following statement runs the **date** program:

```
syscmd(date)
```

Using Unique File Names

Use the built-in **maketemp** to make a unique file name from a program. If this macro receives an argument that contains the string XXXXX, it changes the XXXXX to the process ID of the current process. For example, for the statement:

```
maketemp(myfileXXXXX)
```

the **m4** program returns a string that is `myfile` concatenated with the process ID. Use this string to name a temporary file.

Using Conditional Expressions

The built-in **ifelse** performs conditional testing. In the simplest form:

```
ifelse(a, b, c, d)
```

compares the two strings a and b. If a and b are identical, **ifelse** returns the string c. If they are not identical, it returns string d. For example, you can define a macro called **compare** to compare two strings and return yes if they are the same, or no if they are different, as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

The quote characters prevent the evaluation of ifelse from occurring too early. If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can have any number of arguments, and therefore, provides a limited form of multiple path decision capability. For example:

```
ifelse(a, b, c, d, e, f, g)
```

This statement is logically the same as the following fragment:

```
if(a == b) x = c;
else if(d == e) x = f;
else  x = g;
return(x);
```

If the final argument is omitted, the result is null, so:

```
ifelse(a, b, c)
```

is c if a matches b, and null otherwise.

Manipulating Strings

The built-in **len** returns the byte length of the string that makes up its argument. Thus:

```
len(abcdef)
```

is 6, and:

```
len((a,b))
```

is 5.

The built-in **dlen** returns the length of the displayable characters in a string. Characters made up from 2-byte codes are displayed as one character. Thus, if the string contains any 2-byte international character support characters, the results of **dlen** will differ from the results of **len**.

The built-in **substr** provides substrings of strings. Using input, **substr(*s*, *i*, *n*)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. For example, the function:

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time
```

The built-in **index(s1, s2)** returns the index (position) in **s1** where the string **s2** occurs, or -1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration. It has the general form:

```
translit(s, f, t)
```

which modifies `s` by replacing any character found in `f` by the corresponding character of `t`. For example, the function:

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If `t` is shorter than `f`, characters that do not have an entry in `t` are deleted. If `t` is not present at all, characters from `f` are deleted from `s`. So:

```
translit(s, aeiou)
```

deletes vowels from string S.

The built-in **dnl** deletes all characters that follow it up to and including the next new line. Use this macro to get rid of empty lines. For example, the function:

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new-line at the end of each line that is not part of the definition. These new-line characters are passed to the output. To get rid of the new-lines, add the built-in **dnl** to each of the lines.

```
define(N, 100) dnl
define(M, 200) dnl
define(L, 300) dnl
```

Printing

The built-in **errprint** writes its arguments on the standard error file. For example:

```
errprint ('error')
```

The built-in **dumpdef** dumps the current names and definitions of items named as arguments. If you do not supply arguments, **dumpdef** prints all current names and definitions. Do not forget to quote the names.

Chapter 13. Creating an Input Language

CONTENTS

About This Chapter 13-3
Writing a Lexical Analyzer Program with lex 13-4
 What lex Does 13-4
 How the Lexical Analyzer Works 13-5
The lex Specification File 13-6
 Rules 13-6
Regular Expressions 13-8
 Putting Blanks in an Expression 13-11
 Other Special Characters 13-11
 Character Classes 13-11
 Matching Rules 13-12
Actions 13-14
 Null Action 13-14
 Same as Next Action 13-14
 Printing a Matched String 13-15
 Finding the Length of a Matched String 13-15
 Getting More Input 13-15
 Putting Characters Back 13-16
 Input/Output Routines 13-17
 Character Set 13-18
 End of File Processing 13-18
Passing Code to the Generated Program 13-19
Defining Substitution Strings 13-20
Start Conditions 13-21
Compiling the Lexical Analyzer 13-22
Using lex with yacc 13-23
Creating a Parser with yacc 13-25
Grammar File 13-26
 main and yyerror 13-26
 yylex 13-27
Using the Grammar File 13-28
 Using Comments 13-28
 Using Literal Strings 13-29
 How to Format the Grammar File 13-29
 Using Recursion in a Grammar File 13-30
 Errors in the Grammar File 13-30
Declarations 13-31

Defining Global Variables 13-32
Start Conditions 13-32
Token Numbers 13-32
Rules 13-34
Repeating Nonterminal Names 13-34
Empty String 13-34
End of Input Marker 13-35
Actions 13-36
Passing Values Between Actions 13-37
Putting Actions in the Middle of Rules 13-38
Programs 13-39
Error Handling 13-40
Providing for Error Correcting 13-41
Clearing the Look Ahead Token 13-42
Lexical Analysis 13-43
Parser Operation 13-44
Shift 13-45
Reduce 13-45
Using Ambiguous Rules 13-47
Understanding Parser Conflicts 13-47
How the Parser Responds to Conflicts 13-49
Turning on Debug Mode 13-50
Creating a Simple Calculator Program - Example 13-51
Compiling the Example Program 13-51
The Parser Source Code 13-52
The Lexical Analyzer Source Code 13-57

About This Chapter

If a program needs to receive input, either interactively or in a batch environment, you must provide a program or routine to receive the input. If the input is complicated, it may require additional code to break the input into pieces that mean something to the program. This section describes two programs that help develop these input programs.

First this chapter describes **lex**, a program that generates a program from a set of rules. The **lex** program generates a program, called a ***lexical analyzer***, that analyzes input and breaks it into categories, such as: numbers, letters or operators. Define the categories in the input to **lex**.

Next the chapter describes the **yacc** program. This program also generates a program from a set of rules. However, the program that **yacc** generates is a ***parser*** program. A parser is a program that analyzes input, using the categories that the lexical analyzer identified, and determines what to do with the input.

Finally, this chapter includes an example of a **lex** and a **yacc** program that together generate a third program, a simple desk calculator.

Writing a Lexical Analyzer Program with lex

The **lex** program helps write a C language program that can receive a character stream input and translate that input into program actions. To use the **lex** program, write a specification file that contains the following parts:

Regular expressions
: Character patterns that the generated lexical analyzer recognizes.

Action statements
: C language program fragments that define how the generated lexical analyzer reacts to regular expressions that it recognizes.

The actual format and logic allowed in this file is discussed in "The lex Specification File" on page 13-6.

What lex Does

Using the information in the specification file, the **lex** program generates a C language program to analyze an input stream according to the specifications. The **lex** program puts the output program in a file called **yy.lex.c**. If the output program recognizes a simple one-word input structure, compile the **yy.lex.c** output file using the command:

```
cc -ll yy.lex.c
```

to get an executable lexical analyzer. However, if the lexical analyzer recognizes more than one-word syntax, create a parser to ensure proper handling of the input (see "Creating a Parser with yacc" on page 13-25).

The **yy.lex.c** output file can be moved to other systems that have a C compiler that supports the **lex** library functions.

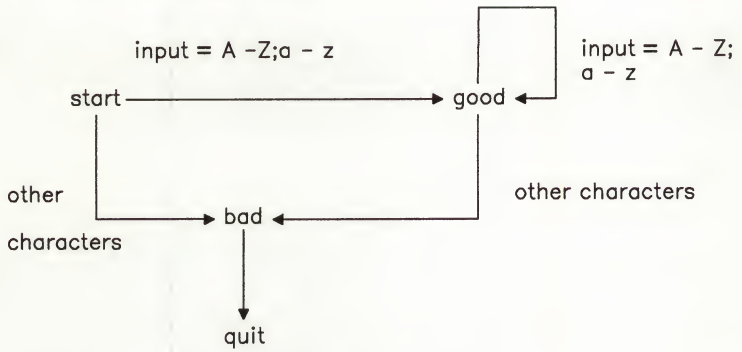
The compiled lexical analyzer performs the following functions:

- Reads an input stream of characters
- Copies the input stream to an output stream
- Breaks the input stream into smaller strings that match the regular expressions in the **lex** specification file
- Executes an action for each regular expression that it recognizes. The action(s) are C language program fragments in the **lex** specification file. The action fragments can call actions or subroutines outside of the action fragment.

How the Lexical Analyzer Works

The lexical analyzer that **lex** generates uses an analysis method called a ***deterministic finite-state automaton***. This method provides for a limited number of conditions that the lexical analyzer can exist in, along with the rules that determine what state the lexical analyzer is in.

For a simple example, Figure 13-1 shows a chart of a program that has three states: start, good and bad. The program gets a stream of characters for input. It begins in the `start` condition. When it receives the first character, the program compares the character with the rule. If the character is alphabetic (according to the rule), the program changes to the `good` state; if it is not alphabetic, the program changes to the `bad` state. The program stays in `good` until it finds a character that does not match its conditions, and then it moves to `bad` (which terminates the program).



A5AC6023

Figure 13-1. Simple Finite State Model

The automaton allows the generated lexical analyzer to look ahead in an input stream more than one or two characters. For example, define two rules in the **lex** specification file, one that looks for the string `ab` and the other that looks for the string `abcdefg`. If the lexical analyzer gets an input string of `abcdefh`, it reads characters to the end of string `abcdefg` before finding that the input string does not match the string `abcdefg`. The lexical analyzer then returns to the rule that looks for the string `ab`, decides that it matches part of the input, and begins trying to find another match using the remaining input `cdefh`.

The lex Specification File

The format of the **lex** specification file is:

```
{definitions}
% %
{rules}
% %
{user subroutines}
```

You can omit the definitions and the user subroutines. The second % % is optional, but the first % % is required to mark the beginning of the rules. The minimum **lex** specification file contains no definitions and no rules:

```
% %
```

Without a specified action for a pattern match, the lexical analyzer copies the input pattern to the output without changing it. Therefore, the previous specification file results in a lexical analyzer that copies all input to the output unchanged.

Rules

The rules section of the specification file contains control decisions that define the lexical analyzer that **lex** generates. The rules are in the form of a table. The left column of the table contains regular expressions; the right column of the table contains ***actions***. Actions are C language program fragments. When the lexical analyzer finds a match for the regular expression that appears in the left column of the table, the lexical analyzer executes the action.

For example, to create a lexical analyzer to look for the string `integer` and print a message when the lexical analyzer finds the string, define a rule:

```
integer      printf("found keyword int");
```

This example uses the C language library function **printf** to print the string. The first blank or tab character in the action indicates the end of the expression. When using only one expression in an action, put it on the same line and to the right of the regular expression (`integer`). When using more than one statement, or if the statement takes more than one line, enclose the action in braces, the same as in a C language program.

For another example, a lexical analyzer to change some words in a file from British spelling to the American spelling has a specification file that contains rules such as:

```
colour                  printf("color");
mechanise               printf("mechanize");
petrol                  printf("gas");
```

This specification file is not complete because it changes the word `petroleum` to `gaseum`.

Regular Expressions

Specifying ***regular expressions*** in a **lex** specification file is similar to methods used in **sed** or **ed**. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. Text characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features.

The letters of the alphabet and the digits are always text characters. For example, the regular expression `integer` matches the string `integer`, and the expression `a57D` looks for the string `a57D`.

Operators

The operator characters for specifying a regular expression are:

Symbol	Use
"	Encloses literal strings to interpret as text characters
\	(Escape character). Indicates that the operator symbol represents the character rather than the operator when used before one of the character class operators. For example, `[\^abc]` represents the class of characters that includes the characters `^abc`.
[]	Encloses character classes.
^	In a character class, indicates the complement of the set of characters when the `^` is the first character in a set of characters. For example, `[^abc]` matches all characters except `a`, `b` or `c`, including all special or control characters. Similarly, `[^a-zA-Z]` is any character that is not a letter. In an expression, indicates a match only when the expression is at the beginning of the line when the `^` is the first character in an expression.

Figure 13-2 (Part 1 of 3). Regular Expression Operators

Symbol	Use
−	In a character class, indicates a range of characters from the ASCII value of the character that comes before the − to the ASCII value of the character that follows the −. For example, `[a-z0-9]` indicates the character class containing all the lowercase letters and the digits. A range can be either in ascending or descending order, but the order is that of the ASCII values of the characters for the RT system. If the program is moved to a system that uses a different set of character codes (like EBCDIC), the range may be a different set of characters. **lex** displays a warning message if moving to another system is likely to cause a problem.
?	(Optional element). Indicates that the character that precedes the ? is not required to match the string, but may be present in that position. For example, `ab?c` matches either `ac` or `abc`
.	Matches any single character except new-line.
*	Matches any number of consecutive occurrences, including zero, of the character that comes before the *. For example, `a*` is any number of consecutive `a` characters, including zero. The usefulness of matching zero occurrences is more obvious in complicated expressions. For example, the expression, `[A-Za-z][A-Za-z0-9]*` indicates all alphanumeric strings with a leading alphabetic character, including strings that are only one alphabetic character. Use this expression for recognizing identifiers in computer languages.
+	Matches any number of consecutive occurrences, but not zero, of the character that comes before the +. For example, `a+` is one or more instances of `a`. Also, `[a-z]+` is all strings of lowercase letters.
\|	Indicates a match for either the expression that precedes the \| or the expression that follows the \|. For example, `ab\|cd` matches either `ab` or `cd`.
()	Groups more complex expressions. For example, `(ab\|cd+)?(ef)*` matches such strings as `abefef`, `efefef`, `cdef`, or `cddd`; but not `abc`, `abcd`, or `abcdef`.
a/b	/ indicates a match of expression `a` only if expression `b` immediately follows expression `a`. For example, `ab/cd` matches the string `ab` but only if followed by `cd`.

Figure 13-2 (Part 2 of 3). Regular Expression Operators

Symbol	Use
$	Indicates a match only when the expression is at the end of the line when used as the last character in an expression. For example, `ab$` is the same as `ab/\n` where \n is a new-line character. See the description of the `a/b` operator.
{ }	When enclosing numbers, the numbers indicate a number of consecutive occurrences of the expression that comes before it. For example, `a{1,5}` indicates a match for from 1 to 5 occurrences of the letter `a`. When enclosing a name, the name represents a string defined earlier in the specification file. Define the named string in the first part of the **lex** specification, before the rules. For example, `{digit}` looks for a defined string named `digit` and inserts it at that point in the expression.
<x>	Encloses a start condition (see "Start Conditions" on page 13-21). The lexical analyzer executes the associated action only if the lexical analyzer is in the indicated start condition (x). If the condition of *being at the beginning of a line* is start condition ONE, then the ^ operator would be the same as the expression, <ONE>.

Figure 13-2 (Part 3 of 3). Regular Expression Operators

To use the operator characters as text characters, indicate that they are text characters by using one of the escape sequences: " " (quotes) or \ (backslash). The operator " (quotation mark) indicates that what is between a pair of quotes is text. Thus:

```
xyz"++"
```

matches the string xyz++. Note that a part of a string may be quoted. Quoting an ordinary text character has no effect. For example, the expression:

```
"xyz++"
```

is the same as the previous one. Quoting all characters that are not letters or numbers, ensures that text is interpreted as text.

Another way to turn an operator character into a text character is to put a backslash character before it. For example:

```
xyz\+\+
```

is another form of the preceding expressions.

Putting Blanks in an Expression

Normally, blanks or tabs end a rule and therefore, the expression that defines a rule. However, you can enclose the blanks or tab characters in quotation marks to include them in the expression. Use quotes around all blanks in expressions that are not already within sets of brackets ([]).

Other Special Characters

The **lex** program recognizes many of the normal C language special characters. These character sequences are:

Sequence	Meaning
\n	New-line - Do not use the actual new-line character in an expression.
\t	Tab
\b	Backspace
\\	Backslash

Figure 13-3. Special Characters

When using these special characters in an expression, you do not need to enclose them in quotes. Every character, except these special characters and the previously described operator symbols, is always a text character.

Character Classes

Character classes are ranges of characters that **lex** uses to match a single character in the input stream. For example, a character class may contain the letters `a, b, c`. If this character class is a match pattern, **lex** accepts any one of the characters `a, b` or `c` from the input stream.

Define character classes using the [] operator pair. Therefore, to define the above character class use the following expression:

```
[abc]
```

The operator symbols, `-`, `^` and `\` can also help define the patterns represented in a character class. See "Operators" on page 13-8 for the definitions of these symbols. All other operators within square brackets do not have any meaning other than as an ordinary character.

Matching Rules

When more than one expression can match the current input, **lex** chooses in the following order:

1. The longest match
2. Among rules that match the same number of characters, the rule that occurs first.

For example, if the rules

```
integer      keyword action...;
[a-z]+       identifier action...;
```

are given in that order, and `integers` is the input word, **lex** matches the input as an identifier, because `[a-z]` matches eight characters while `integer` matches only seven. However, if the input is `integer`, both rules match seven characters. **lex** selects the keyword rule because it occurs first. A shorter input, such as `int`, does not match the expression `integer` and so **lex** selects the identifier rule.

Matching a String Using Wildcard Characters

Because **lex** chooses the longest match first, do not use rules containing expressions like .*. For example:

```
'.*'
```

might seem like a good way to recognize a string in single quotes. However, the lexical analyzer reads far ahead, looking for a distant single quote to complete the long match. If a lexical analyzer with such a rule gets the following input:

```
'first' quoted string here, 'second' here
```

it matches:

```
'first' quoted string here, 'second'
```

To find the smaller strings, `first` and `second`, use the following rule:

```
'[^'\n]*'
```

This rule stops after `'first'`.

Errors of this type are not far reaching, because the . (period) operator does not match a new-line character. Therefore, expressions like .* stop on the current line. Do not try to defeat this with expressions like [.\n] +. The lexical analyzer tries to read the entire input file and an internal buffer overflow occurs.

Finding Strings within Strings

The **lex** program partitions the input stream, and does not search for all possible matches of each expression. Each character is accounted for once and only once. For example, to count occurrences of both she and he in an input text, try the following rules:

```
she                     s++
he                      h++
\n                      |
.                       ;
```

where the last two rules ignore everything besides he and she. However, because she includes he, **lex** does *not* recognize the instances of he that are included in she.

To override this choice, use the action REJECT. This directive tells **lex** to go to the next rule. **lex** then adjusts the position of the input pointer to where it was before the first rule was executed, and executes the second choice rule. For example, to count the included instances of he, use the following rules:

```
she                     {s++; REJECT;}
he                      {h++; REJECT;}
\n                      |
.                       ;
```

After counting the occurrences of she, **lex** rejects the input stream and then counts the occurrences of he. Because in this case she includes he but not vice versa, and you can omit the REJECT action on he. In other cases, it may be difficult to determine which input characters are in both classes.

In general, REJECT is useful whenever the purpose of **lex** is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other.

Actions

When the lexical analyzer matches one of the regular expressions in the rules section of the specification file, it executes the ***action*** that corresponds to the regular expression. Without rules to match all strings in the input stream, the lexical analyzer copies the input to standard output. Therefore, do not create a rule that only copies the input to the output. Use this default output to find gaps in the rules.

When using **lex** to process input for a parser that **yacc** produces, provide rules to match all input strings. Those rules must generate output that **yacc** can interpret.

Null Action

To ignore the input associated with a regular expression, use a ; (C language null statement) as an action. For example:

```
[ \t\n]   ;
```

ignores the three spacing characters (blank, tab, and new-line).

Same as Next Action

To avoid repeatedly writing the same action, use the | (vertical bar) character . This character indicates that the action for this rule is the same as the action for the next rule. For example, the example to ignore blank, tab and new-line characters (shown above), can be written as:

```
"    "                    |
" \t"                     |
" \n"                      ;
```

The quotation marks around \n and \t are not required.

Printing a Matched String

To find out what text matched an expression in the rules section of the specification file, include a C language **printf** function as one of the actions for that expression. When the lexical analyzer finds a match in the input stream, the program puts that matched string in an external character array, called **yytext**. To print the matched string, use a rule like:

```
[a-z]+              printf("%s",yytext);
```

The C language function **printf** accepts a format argument and data to be printed. In this example the arguments to **printf** have the following meanings:

%s A symbol that converts the data to type **string** before printing.
yytext The name of the array containing the data to be printed

Printing the output like this is common. You may want to define it as a macro in the definitions section of the specification file. If this action is defined as ECHO, then the rules section entry looks like:

```
[a-z]+              ECHO;
```

Finding the Length of a Matched String

To find the number of characters that the lexical analyzer matched for a particular regular expression, use the external variable **yyleng**. For example, to count both the number of words and the number of characters in words in the input, use the following action:

```
[a-zA-Z]+       {words++;chars += yyleng;}
```

This action totals the number of characters in the words matched and puts that number in chars.

The following expression finds the last character in the string matched:

```
yytext[yyleng-1]
```

Getting More Input

The lexical analyzer may run out of input before it completely matches an expression in a rules file. In this case, include a call to the **lex** function **yymore** in the action for that rule. Normally, the next string from the input stream overwrites the current entry in **yytext**. If you use **yymore**, the next string from the input stream is added to the end of the current entry in **yytext**.

For example, to define a language that includes the following syntax:

- A string is any set of characters between " (quotation marks)
- A \ (backslash) must come before all strings.

use the rules:

```
\"[^"]*  {
                if (yytext[yyleng-1] =='\\')
                yymore();
         else
                ... normal user processing
                }
```

When this lexical analyzer receives a string such as `"abc\"def"`, it first matches the five characters `"abc\`. Then the call to **yymore** adds the next part of the string `"def` to the end. The part of the action code labeled `normal processing` must process the final quote that ends the string.

Putting Characters Back

The lexical analyzer may not need all of the characters that are matched by the currently successful expression, or it may need to return matched characters to the input stream to be checked again for another match. To return characters to the input stream, use the call:

```
yyless(n)
```

where `n` is the number of characters of the current string to keep. Characters that are beyond the `nth` character in the stream are returned to the input stream. This function provides the same type of look ahead that the / operator uses, but it allows more control over its usage.

Use the **yyless** function to process text more than once. For example, when parsing a C language program an expression such as `x=-a` is difficult to understand. Does it mean *x is equal to minus a*, or is it an older representation of *x -= a* which means *decrement x by the value of a*? To treat this expression as *x is equal to minus a*, but print a warning message, use a rule such as:

```
=-[a-zA-Z]  {
            printf("Operator (=-) ambiguous\n");
            yyless(yyleng-1);
            ... action for =-...
            }
```

Input/Output Routines

The **lex** program allows a program to use the input/output (I/O) routines it uses. These routines are:

input Returns the next input character

output(c) Writes the character `c` on the output

unput(c) Pushes the character `c` back onto the input stream to be read later by **input**.

lex provides these routines as macro definitions. You can override them and provide other versions.

These routines define the relationship between external files and internal characters. If you change them, change them all in the same way and they should follow these rules:

- All routines must use the same character set.
- The **input** routine must return a value of zero to indicate end of file.
- Do not change the relationship of **unput** to **input** or the look ahead functions will not work.

The standard **lex** library allows the lexical analyzer to back up a maximum of 100 characters.

Create a different version of **input** to be able to read a file containing nulls. Using the normal version of **input**, the returned value of 0 (from the null characters) indicates the end of file and ends the input.

Character Set

The lexical analyzers that **lex** generates process character I/O through the routines **input, output,** and **unput**. Therefore, to return values in **yytext**, **lex** uses the character representation that these routines use. Internally however, **lex** represents each character with a small integer. When using the standard library, this integer is the value of the bit pattern that the computer uses to represent the character. Normally, the letter **a** is represented in the same form as the character constant **'a'**. If you change this interpretation with different I/O routines, put a translation table in the definitions section of the specification file. The translation table begins and ends with lines that contain only the entries:

```
%T
```

The translation table contains lines of the form:

```
%T
{integer}    {character string}
{integer}    {character string}
{integer}    {character string}
%T
```

that indicate the value associated with each character.

End of File Processing

When the lexical analyzer reaches the end of a file, it calls a library routine called **yywrap**. This routine returns a value of 1 to indicate to the lexical analyzer that it should continue with normal wrap-up at the end of input. However, if the lexical analyzer receives input from more than one source, change the **yywrap** function. The new function must get the new input and return a value of 0 to the lexical analyzer. A return value of 0 indicates that the program should continue processing.

You can also include code to print summary reports and tables when the lexical analyzer ends in a new version of **yywrap**. The **yywrap** function is the only way to force **yylex** to recognize the end of input.

Passing Code to the Generated Program

You can define variables in either the definitions section or the rules section of the specification file. **lex** changes statements in the specification file into a lexical analyzer. Any line in the specification file that **lex** cannot interpret is passed, unchanged, to the lexical analyzer. Three types of entries can be passed to the lexical analyzer in this manner:

- *Lines beginning with a blank or tab* that are not a part of a **lex** rule are copied into the lexical analyzer. If this entry occurs before the first %% in the specification file, the entry is external to any function in the code. If the entry occurs after the first %%, it must be a C language program fragment that defines a variable. Define these statements before the first **lex** rule in the specification file.
- *Lines beginning with a blank or tab* that are program comments are included as comments in the generated lexical analyzer. The comments must be in the C language format for comments.
- *Any lines that lie between lines containing only %{ and %}* is copied to the lexical analyzer. The symbols %{ and %} are not copied. Use this format to enter preprocessor statements that must begin in column 1, or to copy lines that do not look like program statements.
- *Any lines occurring after the third %% delimiter* are copied to the lexical analyzer without format restrictions.

Defining Substitution Strings

You can define string macros that **lex** expands when it generates the lexical analyzer. Define them before the first %% delimiter in the **lex** specification file. Any line in this section that begins in column 1 and that does not lie between %{ and %} defines a **lex** substitution string. Substitution string definitions have the general format:

```
name                    translation
```

where `name` and `translation` are separated by a least one blank or tab, and `name` begins with a letter. When **lex** finds the string `name` enclosed in { } (braces) in the rules part of the specification file, it changes `name` to the string defined in `translation` and deletes the braces.

For example, to define the names `D` and `E`, put the following definitions before the first %% delimiter in the specification file:

```
D           [0-9]
E           [DEde][-+]{D}+
```

Then, use these names in the rules section of the specification file to make the rules shorter:

```
{D}+                    printf("integer");
{D}+ "." {D}*({E})?     |
{D}*"."{D}+({E})?       |
{D}+{E}                 printf("real");
```

You can also include the following items in the definitions section:

- Character set table
- List of start conditions
- Changes to size of arrays to accommodate larger source programs.

Start Conditions

Any rule may be associated with a start condition. **lex** recognizes that rule only when **lex** is in that start condition. You can change the current start condition at any time.

Define start conditions in the definitions section of the specification file by using a line in the following form:

```
% Start  name1 name2
```

The symbols, `name1` and `name2`, are names that represent conditions. There is no limit to the number of conditions and they can appear in any order. You can also shorten the word `Start` to `s` or `S`.

When using a start condition in the rules section of the specification file, enclose the name of the start condition in <> (angle brackets) at the beginning of the rule:

`<name1>` *expression*

defines a rule, *expression* that **lex** recognizes only when **lex** is in start condition `name1`. To put **lex** in a particular start condition, execute the action statement (in the action part of a rule):

```
BEGIN name1;
```

This statement changes the start condition to `name1`. To resume the normal state:

```
BEGIN 0;
```

resets **lex** to its initial condition. A rule can be active in several start conditions. For example:

```
<name1,name2,name3>
```

is a legal prefix. Any rule that does not begin with a start condition is always active.

Compiling the Lexical Analyzer

Compiling a **lex** program is a two-step process:

1. Use **lex** to change the specification file into a C language program. The resulting program is in the **lex.yy.c** file.
2. Use the **cc -ll** command to compile and link the program with a library of **lex** subroutines. The resulting executable program is in the **a.out** file.

For example, if the **lex** specification file is called **lextest**, enter the following commands:

```
lex lextest
cc lex.yy.c -ll
```

Although the default **lex** I/O routines use the C language standard library, the lexical analyzers that **lex** generates do not. You can include different copies of the **input, output,** and **unput** routines to avoid using the library (see "Input/Output Routines" on page 13-17).

Using lex with yacc

When used alone, the **lex** program generator makes a lexical analyzer that recognizes simple one-word input or receives statistical input. You can also use **lex** with a parser generator, such as **yacc**. The **yacc** program generates a program, called a parser, that analyzes the construction of more than one word input. This parser program operates well with lexical analyzers that **lex** generates. The **lex** program recognizes only regular expressions and formats them into character packages called ***tokens***.

token The smallest independent unit of meaning as defined by either the parser or the lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax.

yacc produces parsers that recognize many types of grammar with no regard to context. These parsers need a preprocessor to recognize input tokens such as the preprocessor that **lex** produces.

When using **lex** to make a lexical analyzer for a parser, the lexical analyzer (from **lex**) partitions the input stream. The parser (from **yacc**) assigns structure to the resulting pieces. Figure 13-4 on page 13-24 shows how the two generated programs work together. You can also use other programs along with the programs generated by either **lex** or **yacc**.

The **yacc** program must have a lexical analyzer named **yylex**, which is what the lexical analyzer from **lex** is named. Normally, the default main program in the **lex** library calls this routine, but if **yacc** is loaded and its main program is used, **yacc** calls **yylex**. In this case, each **lex** rule should end with:

```
return(token);
```

where the appropriate token value is returned.

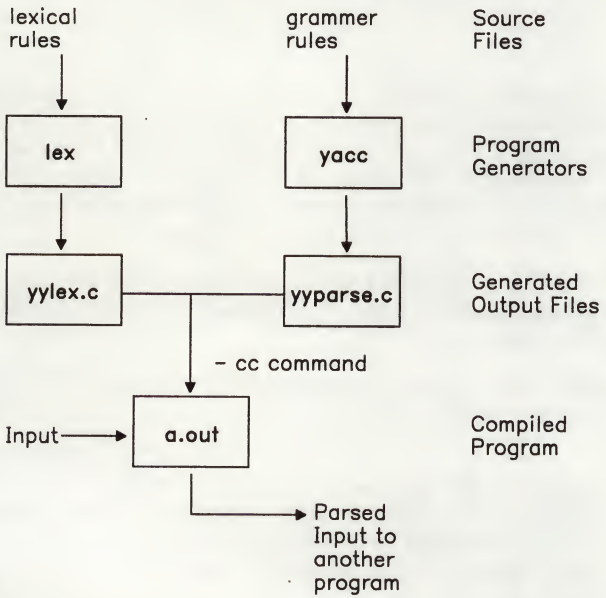
To find out the names for tokens that **yacc** uses, compile the **lex** output file as part of the **yacc** output file by placing the line:

```
#include "lex.yy.c"
```

in the last section of the **yacc** grammar file. For example, if the grammar file is **good** and the specification file is **better**, the final program is created with the following command sequence:

```
yacc good
lex better
cc y.tab.c -ly -ll
```

The **yacc** library (`-ly` in the preceding example) should be loaded before the **lex** library to get a main program that invokes the **yacc** parser. You can generate **lex** and **yacc** programs in either order.



A5AC6024

Figure 13-4. lex with yacc

Creating a Parser with yacc

The **yacc** program creates parsers that define and enforce structure for character input to a computer program. To use this program, supply the following inputs:

Grammar file
: A source file that contains the specifications for the language to recognize. This file also contains the programs **main**, **yyerror** and **yylex**. You must supply these programs.

main
: A C language program that as a minimum contains a call to the function **yyparse** that **yacc** generates. A limited form of this program is in the **yacc** library.

yyerror
: A C language program to handle errors that can occur during parser operation. A limited form of this program is in the **yacc** library.

yylex
: A C language program to perform lexical analysis on the input stream and pass tokens to the parser. You can generate this lexical analyzer using the **lex** program.

When **yacc** gets a specification, it generates a file of C language programs, called **y.tab.c**. When compiled using the **cc** command, these programs form a function called **yyparse** that returns an integer. When it is called, **yyparse** calls **yylex**, the lexical analyzer to get input tokens. **yylex** continues providing input until either the parser detects an error, or **yylex** returns an end-marker token to indicate the end of operation. If an error occurs and **yyparse** cannot recover, it returns a value of 1 to **main**. If it finds the end-marker token, **yyparse** returns a value of 0 to **main**.

Grammar File

To use **yacc** to generate a parser, give it a grammar file that describes the input data stream and what the parser is to do with the data. The grammar file includes rules describing the input structure, code to be invoked when these rules are recognized, and a routine to do the basic input.

The **yacc** program uses the information in the grammar file to generate a program that controls the input process. This program, called a ***parser***, calls an input routine (the ***lexical analyzer***) to pick up the basic items (called ***tokens***) from the input stream. The parser organizes these tokens according to the structure rules in the grammar file. The structure rules are called ***grammar rules***. When the parser recognizes one of these rules, it executes the user code supplied for that rule. The user code is called an ***action***. Actions return values and use the values returned by other actions.

Use the C programming language to write the action code and other subroutines. **yacc** uses many of the C language syntax conventions for the grammar file.

main and yyerror

You must provide these two routines for the parser. To ease the initial effort of using **yacc**, the **yacc** library contains simple versions of **main** and **yyerror**. Include these routines using the **−ly** argument to the loader (or to the **cc** command). The source code for the **main** library program is:

```
main()
{
     return ( yyparse() );
}
```

The source code for the **yyerror** library program is:

```
#include <stdio.h>

yyerror(s)
        char *s;
{
        fprintf( stderr, " %s\n" ,s);
}
```

The argument to **yyerror** is a string containing an error message, usually the string `syntax error`.

These are very limited programs. You should provide more function in these routines. For example, keep track of the input line number and print it along with the message when a syntax error is detected. You may also want to use the value in the external integer variable **yychar**. This variable contains the look-ahead token number at the time the error was detected.

yylex

The input routine that you supply must be able to:

- Read the input stream
- Recognize basic patterns in the input stream
- Pass the patterns to the parser along with ***tokens*** that define the pattern to the parser.

A ***token*** is a symbol or name that tells the parser which pattern is being sent to it by the input routine. A ***nonterminal symbol*** is the structure that the parser recognizes.

For example, if the input routine separates an input stream into the tokens of WORD, NUMBER and PUNCTUATION, and it receives the input:

```
I have 9 turkeys.
```

the program could choose to pass the following strings and tokens to the parser:

String	Token
I	WORD
have	WORD
9	NUMBER
turkeys	WORD
.	PUNCTUATION

The parser must contain definitions for the tokens that the input routine passes to it. If you use the **-d** option for **yacc**, it generates a list of tokens in a file called **y.tab.h**. This list is a set of **#define** statements that allow the lexical analyzer (**yylex**) to use the same tokens as the parser.

To avoid conflict with the parser, do not use names that begin with the letters yy.

You can use **lex** to generate the input routine, or you can write it in the C language. See "The lex Specification File" on page 13-6 for information about using **lex**.

Using the Grammar File

A **yacc** grammar file consists of three sections:

- Declarations
- Rules
- Programs.

Two %% (percent signs) that appear together separate the sections of the grammar file. To make the file easier to read, put the %% on a line by themselves. A complete grammar file looks like:

```
declarations
% %
rules
% %
programs
```

The declarations section may be empty. If you omit the programs section, omit the second set of %%. Therefore, the smallest **yacc** grammar file is:

```
% %
rules
```

yacc ignores blanks, tabs and new-line characters in the grammar file. Therefore, use these characters to make the grammar file easier to read. Do not, however, use blanks, tabs or new-lines in names or reserved symbols.

Using Comments

Put comments in the grammar file to explain what the program is doing. You can put comments anywhere in the grammar file that you can put a name. However, to make the file easier to read, put the comments on lines by themselves at the beginning of functional blocks of rules. A comment in a **yacc** grammar file looks the same as a comment in a C language program; it is enclosed in /* */. For example:

```
/* This is a comment on a line by itself. */
```

Using Literal Strings

A literal string is one or more characters enclosed in '' (single quotes). As in the C language, the \ (backslash) is an escape character within literals, and all the C language escape codes are recognized. Thus, **yacc** accepts the symbols in the following table:

Symbol	Definition
'\n'	New-line
'\r'	Return
'\''	Single quote (')
'\\'	Backslash (\)
'\t'	Tab
'\b'	Backspace
'\f'	Form feed
'\xxx'	The value xxx in octal

Figure 13-5. yacc Literal Strings

Never use \0 or 0 (the NUL character) in grammar rules.

How to Format the Grammar File

Use the following guidelines to help make the **yacc** grammar file more readable:

1. Use uppercase letters for token names and lowercase letters for nonterminal symbol names.
2. Put grammar rules and actions on separate lines to allow changing either one without changing the other.
3. Put all rules with the same left side together. Enter the left side once and use the vertical bar to begin the rest of the rules for that left side.
4. For each set of rules with the same left side, enter the semicolon once on a line by itself following the last rule for that left side. You can then add new rules easily.
5. Indent rule bodies by two tab stops and action bodies by three tab stops.

Using Recursion in a Grammar File

Recursion is the process of using a function to define itself. In language definitions, these rules normally take the form:

```
rule    :   <end case>
        |   rule,<end case>
```

which means that the simplest case of the `rule` is the `end case`, but `rule` can also be made up of more than one occurrence of `end case`. The entry in the second line that uses `rule` in the definition of `rule` is the recursion. The parser cycles through the input until the stream is reduced to the final `end case`.

When using recursion in a rule, always put the call to the name of the rule as the leftmost entry in the rule (as it is in the preceding example). If the call to the name of the rule occurs later in the line, such as:

```
rule    :   <end case>
        |   <end case>,rule
```

the parser may run out of internal stack space, stopping the parser.

Errors in the Grammar File

The **yacc** program cannot produce a parser for all sets of grammar specifications. If the grammar rules contradict themselves or require different matching techniques than **yacc** has, **yacc** will not produce a parser. In most cases, **yacc** provides messages to indicate the errors. To correct these errors, redesign the rules in the grammar file, or provide a lexical analyzer (input program to the parser) to recognize the patterns that **yacc** cannot.

Declarations

The declarations section of the **yacc** grammar file contains:

- Declarations for any variables or constants used in other parts of the grammar file
- **#include** statements to use other files as part of this file (used for library header files)
- Statements that define processing conditions for the generated parser.

A declaration for a variable or constant follows the syntax of the C programming language:

type specifier declarator ;

where, *type specifier* is a data type keyword, and *declarator* is the name of the variable or constant. Names can be any length and consist of letters, dots, underscores, and digits. A name cannot begin with a digit. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

Without declaring a name in the declarations section, you can use that name only as a nonterminal symbol. Define each nonterminal symbol by using it as the left side of at least one rule in the rules section. The **#include** statements are identical to C language syntax, and perform the same function.

The **yacc** program has a set of keywords that define processing conditions for the generated parser. Each of the keywords begin with a % and is followed by a list of tokens. These keywords are:

%left	Identifies tokens that are left-associative with other tokens
%nonassoc	Identifies tokens that are not associative with other tokens
%right	Identifies tokens that are right-associative with other tokens
%start	Identifies a name for the start symbol
%token	Identifies the token names that **yacc** accepts. Declare all token names in the declarations section.

All of the tokens on the same line have the same precedence level and associativity; the lines appear in the file in order of increasing precedence or binding strength. Thus:

```
%left '+' '-'
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators. The + (plus) and - (minus) are left associative and have lower precedence than * (asterisk) and / (slash), which are also left associative.

Defining Global Variables

To define variables to be used by some or all actions, as well as the lexical analyzer, enclose the declarations for those variables in the %{ %} symbols. Declarations enclosed in these symbols are called ***global*** variables. For example, to make the `var` variable available to all parts of the complete program, use the following entry in the declarations section of the grammar file:

```
%{ int var = 0; %}
```

Start Conditions

The parser recognizes a special symbol called the ***start*** symbol. The start symbol is the name of the rule in the rules section of the grammar file that describes the most general structure of the language to be parsed. Because it is the most general structure, it is the structure where the parser starts in its *top down* analysis of the input stream. Declare the start symbol in the declarations section using the keyword, `%start`. If you do not declare the name of the start symbol, the parser uses the name of the first grammar rule in the grammar file.

For example, when parsing a C language procedure, the most general structure for the parser to recognize is:

```
main()
{
   code_segment
}
```

The start symbol should point to the rule that describes this structure. All remaining rules in the file describe ways to identify lower-level structures within the procedure.

Token Numbers

Token numbers are nonnegative integers that represent the names of tokens. If the lexical analyzer passes the token number to the parser instead of the actual token name, both programs must agree on the numbers assigned to the tokens.

You can assign numbers to the tokens used in the **yacc** grammar file. If you do not assign numbers to the tokens, **yacc** assigns numbers using the following rules:

1. A literal character is the numerical value of the character in the ASCII character set.
2. Other names are assigned token numbers starting at 257.

Note: Do not assign a token number of 0. This number is assigned to the ***endmarker*** token. You cannot redefine it.

To assign a number to a token (including literals) in the declarations section of the grammar file, put a positive integer (not zero) immediately following the token name in the `%token` line. This integer is the token number of the name or literal. Each number must be different from the rest of the token numbers. All lexical analyzers used with **yacc** must return a 0, or a negative value for a token when they reach the end of their input.

Rules

The ***rules*** section contains one or more grammar rules. Each rule describes a structure and gives it a name. A grammar rule has the form:

```
A : BODY;
```

where A is a nonterminal name, and BODY is a sequence of zero or more names and literals. The colon and the semicolon are required **yacc** punctuation.

Repeating Nonterminal Names

If there are several grammar rules with the same nonterminal name, use the | (vertical bar) to avoid rewriting the left side. In addition, use the ; (semicolon) only at the end of all rules joined by vertical bars. Thus the grammar rules:

```
A  :  B  C  D  ;
A  :  E  F  ;
A  :  G  ;
```

can be given to **yacc** as:

```
A  :  B  C  D
   |  E  F
   |  G
   ;
```

by using the vertical bar.

Empty String

To indicate a nonterminal symbol that matches the empty string, use a ; (semicolon) by itself in the body of the rule. Therefore, to define a symbol empty that matches the empty string, use a rule like the following rule:

```
empty  :  ;
```

End of Input Marker

When the lexical analyzer reaches the end of the input stream, it sends an end of input marker to the parser. This marker is a special token called ***endmarker***, and has a token value of 0. When the parser receives an end of input marker, it checks to see that it has assigned all of the input to defined grammar rules, and that the processed input forms a complete unit (as defined in the **yacc** grammar file). If the input is a complete unit, the parser stops. If the input is not a complete unit, the parser signals an error and stops.

The lexical analyzer must send the end of input marker at the correct time, such as the end of a file, or the end of a record.

Actions

With each grammar rule, you can specify ***actions*** to be performed each time the parser recognizes the rule in the input stream. Actions return values and obtain the values returned by previous actions. The lexical analyzer can also return values for tokens.

An action is a C language statement that does input and output, calls subprograms, and alters external vectors and variables. Specify an action in the grammar file with one or more statements enclosed in braces { and }. For example:

```
A  :  '('B')'
   {
     hello(1, "abc" );
   }
```

and

```
XXX  :  YYY  ZZZ
     {
     printf("a message\n");
     flag = 25;
     }
```

are grammar rules with actions.

Passing Values Between Actions

An action can get values generated by other actions by using the **yacc** parameter keywords that begin with a dollar sign ($1, $2 ...). The keywords that begin with a dollar sign refer to the values returned by the components of the right side of a rule, reading from left to right. For example, if the rule is:

```
A  :  B  C  D  ;
```

then $2 has the value returned by the rule that recognized C, and $3 the value returned by the rule that recognized D.

To return a value, the action sets the pseudo-variable $$ to some value. For example, the action:

```
{ $$ = 1;}
```

returns a value of one.

By default, the value of a rule is the value of the first element in it ($1). Therefore, you do not need to provide an action for rules that have the following form:

```
A : B ;
```

Putting Actions in the Middle of Rules

To get control of the parsing process before a rule is completed, write an action in the middle of a rule. If this rule returns a value through the $ parameters, actions that come after it can use that value. It can use values returned by actions that come before it. Therefore, the rule:

```
A   :  B
          {
             $$ =1;
          }
          C
   {
          x = $2;
          y = $3;
   }
   ;
```

sets x to 1 and y to the value returned by C.

Internally, **yacc** creates a new nonterminal symbol name for the action that occurs in the middle, and it creates a new rule matching this name to the empty string. Therefore, **yacc** treats the preceding program as if it were written in the following form:

```
$ACT   :   /* empty */
       {
          $$ = 1;
       }
       ;
A      :   B   $ACT   C
       {
          x = $2;
          y = $3;
       }
       ;
```

where $ACT is an empty action.

Programs

The programs section contains C language programs that perform functions used by the actions in rules section. In addition, if you write a lexical analyzer (input routine to the parser), include it in the programs section.

Error Handling

When the parser reads an input stream, that input stream might not match the rules in the grammar file. The parser detects the problem as early as possible. If there is an error handling routine in the grammar file, the parser can allow for entering the data again, skipping over the bad data, or a cleanup and recovery action. When the parser finds an error, for example, it may need to reclaim parse tree storage, delete or alter symbol table entries, and set switches to avoid generating any further output.

When an error occurs, the parser stops unless you provide error handling routines. To keep processing the input to find more errors, restart the parser at a point in the input stream where the parser can try to recognize more input. One way to restart the parser when an error occurs is to discard some of the tokens following the error, and try to restart the parser at that point in the input stream.

The **yacc** program has a special token name, **error**, to use for error handling. Put this token in the rules file at places that an input error might occur so that you can provide a recovery routine. If an input error occurs in this position, the parser executes the action for the **error** token, rather than the normal action.

To prevent a single error from producing many error messages, the parser remains in error state until it processes three tokens following an error. If another error occurs while the parser is in the error state, the parser discards the input token and does not produce a message.

As an example, a rule of the form:

```
stat  :  error ';'
```

tells the parser that, when there is an error, it should skip over the token and all following tokens until it finds the next semicolon. All tokens after the error and before the next semicolon are discarded. When the parser finds the semicolon, it reduces this rule and performs any cleanup action associated with it.

Providing for Error Correcting

You can also allow the person entering the input stream in an interactive environment to correct any input errors by entering a line in the data stream again. For example:

```
input : error  '\n'
        {
           printf(" Reenter last line: " );
        }
        input
      {
        $$ = $4;
      }
      ;
```

is one way to do this. However, in this example the parser stays in the error state for three input tokens following the error. If the corrected line contains an error in the first three tokens, the parser deletes the tokens and does not give a message. To allow for this condition, use the **yacc** statement:

```
yyerrok;
```

When the parser finds this statement, it leaves the error state and begins processing normally. The error recovery example then becomes:

```
input : error '\n'
        {
           yyerrok;
           printf( "Reenter last line: " );
        }
        input
      {
        $$ = $4
      }
      ;
```

Clearing the Look Ahead Token

The ***look ahead token*** is the next token that the parser examines. When an error occurs, the look ahead token becomes the token at which the error was detected. However, if the error recovery action includes code to find the correct place to start processing again, that code must also change the look ahead token. To clear the look ahead token, include the statement:

```
yyclearin ;
```

in the error recovery action.

Lexical Analysis

You must provide a lexical analyzer to read the input stream and send tokens (with values, if required) to the parser that **yacc** generates. The lexical analyzer is a function called **yylex**. The function must return an integer that represents the kind of token that was read. The integer is called the ***token number***. In addition, if a value is associated with the token, the lexical analyzer must assign that value to the external variable **yylval**.

To build a lexical analyzer that works well with the parser that **yacc** generates, use the **lex** program (see "The lex Specification File" on page 13-6).

Parser Operation

The **yacc** program turns the grammar file into a C language program. That program, when compiled and executed, parses the input according to the grammar specification given.

The parser is a finite state machine with a stack. The parser can read and remember the next input token (called the ***look ahead*** token). The ***current state*** is always the state that is on the top of the stack. The states of the finite state machine are represented by small integers. Initially, the machine is in state 0, the stack contains only 0, and no look ahead token has been read.

The machine can perform one of four actions:

shift n The parser pushes the current state onto the stack, makes `n` the current state, and clears the look ahead token.

reduce r The letter `r` is a rule number. When the parser finds a string defined by rule number `r` in the input stream, the parser replaces that string with the rule number in the output stream.

accept The parser looked at all input, matched it to the grammar specification, and recognized the input as satisfying the highest level structure (defined by the start symbol). This action appears only when the look ahead token is the endmarker and indicates that the parser has successfully done its job.

error The parser cannot continue processing the input stream and still successfully match it with any rule defined in the grammar specification. The input tokens it looked at, together with the look ahead token, cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing.

The parser performs the following actions during one process step:

1. Based on its current state, the parser decides whether it needs a look ahead token to decide the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look ahead token being processed or left alone.

Shift

The **shift** action is the most common action the parser takes. Whenever the parser does a shift, there is always a look ahead token. For example, for the following grammar specification rule:

```
IF shift 34
```

if the parser is in the state that contains this rule and the look ahead token is IF, the parser:

1. Pushes the current state down on the stack
2. Makes state 34 the current state (puts it on the top of the stack)
3. Clears the look ahead token.

Reduce

The ***reduce*** action keeps the stack from growing too large. The parser uses reduce actions after it has matched the right side of a rule with the input stream and is ready to replace the characters in the input stream with the left side of the rule. The parser may have to use the look ahead token to decide if the pattern is a complete match.

Reduce actions are associated with individual grammar rules. Because grammar rules also have small integer numbers, you can easily confuse the meanings of the numbers in the two actions, shift and reduce. For example, the action:

```
. reduce 18
```

refers to grammar rule 18, while the action:

```
IF shift 34
```

refers to state 34.

For example, to reduce the rule:

```
A  :  x y z ;
```

The parser pops off the top three states from the stack. The number of states popped equals the number of symbols on the right side of the rule. These states are the ones put on the stack while recognizing x, y, and z. After popping these states, a state is uncovered which is the state the parser was in before beginning to process the rule (the state that needed to recognize rule A to satisfy its rule). Using this uncovered state and the symbol on the left side of the rule, the parser performs an action called **goto**, which is similar to a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

The **goto** action is different from an ordinary shift of a token. The look ahead token is cleared by a shift but is not affected by a **goto** When the three states are popped, the uncovered state contains an entry such as:

```
A  goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the parser executes the code that you included in the rule before adjusting the stack. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a shift takes place, the external variable **yylval** is copied onto the value stack. After executing the code that you provide, the parser performs the reduction. When the parser performs the **goto** action, it copies the external variable **yyval** onto the value stack. The **yacc** variables that begin with $ refer to the value stack.

Using Ambiguous Rules

A set of grammar rules is ***ambiguous*** if any possible input string can be structured in two or more different ways. For example, the grammar rule:

```
expr : expr '-' expr
```

states a rule that forms an arithmetic expression by putting two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not specify how to structure all complex inputs. For example, if the input is:

```
expr - expr - expr
```

using that rule, a program could structure this input as either ***left associative***:

```
(  expr - expr ) - expr
```

or as ***right associative***:

```
expr - ( expr - expr )
```

and produce different results.

Understanding Parser Conflicts

When the parser tries to handle an ambiguous rule, it can become confused over which of its four actions to perform when processing the input. Two major types of conflicts develop:

shift/reduce conflict
: A rule can be evaluated correctly using either a shift action or a reduce action but the result is different.

reduce/reduce conflict
: A rule can be evaluated correctly using one of two different reduce actions, producing two different actions.

A **shift/shift** conflict is not possible. These conflicts result from a rule that is not as complete as it could be. For example, using the previous ambiguous rule, if the parser receives the input:

```
expr - expr - expr
```

after reading the first three parts the parser has:

```
expr - expr
```

which matches the right side of the preceding grammar rule. The parser can reduce the input by applying this rule. After applying the rule, the input becomes:

```
expr
```

which is the left side of the rule. The parser then reads the final part of the input:

```
- expr
```

and reduces it. This produces a left associative interpretation.

However, the parser can also look ahead in the input stream. If when it receives the first three parts:

```
expr - expr
```

it reads the input stream until it has the next two parts, it then has the following input:

```
expr - expr - expr
```

Applying the rule to the rightmost three parts reduces them to `expr`. The parser then has the expression:

```
expr - expr
```

Reducing the expression once more produces a right associative interpretation.

Therefore, at the point that the parser has read only the first three parts, it can take two legal actions: a shift or a reduction. If the parser has no rule to decide between them, this situation is called a **shift/reduce** conflict.

A similar situation occurs if the parser can choose between two legal reduce actions. That situation is called a **reduce/reduce** conflict.

How the Parser Responds to Conflicts

When shift/reduce or reduce/reduce conflicts occur, **yacc** produces a parser by selecting one of the valid steps wherever it has a choice. If you do not provide a rule that makes the choice, **yacc** uses two rules:

1. In a shift/reduce conflict, do the shift.
2. In a reduce/reduce conflict, reduce by the grammar rule that it can apply at the earliest point in the input stream.

Using actions within rules can cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, using the preceding rules leads to an incorrect parser. For this reason, **yacc** reports the number of shift/reduce and reduce/reduce conflicts that it resolved using its rules.

Turning on Debug Mode

For normal operation, the external integer variable **yydebug** is set to 0. However, if you set it to a value that is not zero, the parser generates a description of:

- The input tokens that it receives
- The actions that it takes for each token

while it is parsing an input stream.

Set this variable in one of two ways:

- Put the C language statement:

  ```
  yydebug = 1;
  ```

 in the declarations section of the **yacc** grammar file.

- Use **sdb** to execute the final parser, and set the variable on or off using **sdb** commands. See Chapter 8, "Debugging Programs" on page 8-1 for information about using **sdb**.

Creating a Simple Calculator Program - Example

This section describes the example programs for **lex** and **yacc** that are in the set of example programs. The **lex** and **yacc** programs together create a simple desk calculator program that performs addition, subtraction, multiplication and division operations. The calculator program also allows you to assign values to variables (each designated by a single lowercase letter) and then use the variables in calculations. The files that contain the program are:

File	Content
calc.lex	The **lex** specification file that defines the lexical analysis rules.
calc.yacc	The **yacc** grammar file that defines the parsing rules, and calls the **yylex** function created by **lex** to provide input.

To use these files, they must be in your current directory. Copy them from the directory **/usr/lib/samples/toolbook** if you have installed the Programming Examples. The remaining text expects that the current directory is the directory that contains the **lex** and **yacc** example program files.

Compiling the Example Program

Perform the following steps, in order, to create the example program using **lex** and **yacc**:

1. Process the **yacc** grammar file using the **-d** option. The **-d** option tells **yacc** to create a file that defines the tokens it uses in addition to the C language source code:

   ```
   yacc -d calc.yacc
   ```

2. Use the **li** command to verify that the following files were created:

 y.tab.c The C language source file that **yacc** created for the parser.

 y.tab.h A header file containing define statements for the tokens used by the parser.

3. Process the **lex** specification file:

   ```
   lex calc.lex
   ```

4. Use the **li** command to verify that the following file was created:

 lex.yy.c The C language source file that **lex** created for the lexical analyzer.

5. Compile and link the two C language source files:

```
cc y.tab.c lex.yy.c
```

6. Use the **li** command to verify that the following files were created:

y.tab.o	The object file for **y.tab.c**.
lex.yy.o	The object file for **lex.yy.c**.
a.out	The executable program file.

You can then run the program directly from **a.out** by entering the command:

```
$ a.out
```

or, you can move the program to a file with a more descriptive name, like in the following example, and then run it:

```
$ mv a.out calculate
$ calculate
```

In either case after you start the program, the cursor moves to the line below the $ (command prompt). Then enter numbers and operators in calculator fashion. After you press the **Enter** key, the program displays the result of the operation. If you assign a value to a variable:

```
m=4 <enter>
_
```

the cursor moves to the next line. You can then use the variable in calculations and it will have the value assigned to it:

```
m+5 <enter>
9
_
```

The Parser Source Code

Figure 13-6 on page 13-53 shows the contents of the file **calc.yacc**. This file has entries in all three of the sections of a **yacc** grammar file: declarations, rules and programs.

```
%{
#include <stdio.h>

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /*supplies precedence for unary minus */


%%                    /* beginning of rules section */

list:                          /*empty */
        |
        list stat'\n'
        |
        list error'\n'
        {
          yyerrok;
        }
        ;
```

Figure 13-6 (Part 1 of 4). yacc Grammar File for Calculator Program - calc.yacc

```
stat:    expr
         {
           printf("%d\n",$1);
         }
         |
         LETTER '=' expr
         {
           regs[$1] = $3;
         }
         ;
expr:    '(' expr ')'
         {
           $$ = $2;
         }
         |
         expr '*' expr
         {
           $$ = $1 * $3;
         }
         |
         expr '/' expr
         {
           $$ = $1 / $3;
         }
         |
         expr '%' expr
         {
           $$ = $1 % $3;
         }
         |
         expr '+' expr
         {
           $$ = $1 + $3;
         }
```

Figure 13-6 (Part 2 of 4). yacc Grammar File for Calculator Program - calc.yacc

```
        |
        expr '-' expr
        {
          $$ = $1 - $3;
        }
        |
        expr '&' expr
        {
          $$ = $1 & $3;
        }
        |
        expr '|' expr
        {
          $$ = $1 | $3;
        }
        |
        '-' expr %prec UMINUS
        {
          $$ = -$2;
        }
        |
        LETTER
        {
          $$ = regs[$1];
        }
        |
        number
        ;
number: DIGIT
        {
          $$ = $1;
          base = ($1==0) ? 8:10;
        }
```

Figure 13-6 (Part 3 of 4). yacc Grammar File for Calculator Program - calc.yacc

```
        |
        number DIGIT
        {
          $$ = base * $1 + $2;
        }
        ;
%%
main()
{
 return(yyparse());
}

yyerror(s)
char *s;
{
  fprintf(stderr," %s\n",s);
}

yywrap()
{
  return(1);
}
```

Figure 13-6 (Part 4 of 4). yacc Grammar File for Calculator Program - calc.yacc

Declarations Section

This section contains entries that perform the following functions:

- Includes standard I/O header file
- Defines global variables
- Defines the rule `list` as the place to start processing
- Defines the tokens used by the parser
- Defines the operators and their precedence.

Rules Section

The rules section defines the rules that parse the input stream.

Programs Section

The programs section contains the following routines. Because these routines are included in this file, you do not need to use the **yacc** library when processing this file.

main() The required main program that calls **yyparse()** to start the program.

yyerror(s) This error handling routine only prints a syntax error message.

yywrap() The wrap-up routine that returns a value of 1 when the end of input occurs.

The Lexical Analyzer Source Code

Figure 13-7 on page 13-58 shows the contents of the file **calc.lex**. This file contains include statements for standard input and output, as well as for the file **y.tab.h**. The **yacc** program generates that file from the **yacc** grammar file information if you use the **-d** flag with the **yacc** command. The file **y.tab.h** contains definitions for the tokens that the parser program uses. In addition, **calc.lex** contains the rules to generate the tokens from the input stream.

```
%{

#include <stdio.h>
#include "y.tab.h"
int c;
extern int yylval;
%}
%%
" "         ;
[a-z]       {
              c = yytext[0];
              yylval = c - 'a';
              return(LETTER);
            }
[0-9]       {
              c = yytext[0];
              yylval = c - '0';
              return(DIGIT);
            }
[^a-z0-9\b]     {
                  c = yytext[0];
                  return(c);
                }
```

Figure 13-7. lex Specification File for Calculator Program - calc.lex

Chapter 14. International Character Support

CONTENTS

About This Chapter 14-2
Introduction 14-3
 Features 14-4
 Programming Language Support 14-6
Kernel Modifications 14-8
 Terminal Maps 14-8
 Code Point Support 14-8
Environment Variables 14-10
Character Display and Handling 14-12
 Conversion between char and NLchar 14-12
 Conversion between ASCII and NLchar 14-12
 Character Information Macros 14-13
 The NLyesno Subroutine 14-13
String Handling 14-15
Intersystem Compatibility 14-16
 Basic Networking Utilities 14-17
Limits to Support 14-20
Collation 14-21
Files and Directories 14-22

About This Chapter

This chapter includes a discussion of international character support, which pervasively affects character and string handling to support the character sets and strings of a number of languages. Accordingly, international character support is described in terms of all the affected areas so you can establish and revise environments for character, string, and time and date handling.

International character support is provided primarily through subroutines, but the kernel, commands, and other utilities are also affected.

Japanese Language Support Information

This chapter includes information about Japanese Language Support. When Japanese Language Support is compared to or contrasted with the basic IBM RT AIX Operating System, the latter is referred to as *AIX*.

The discussion of Japanese Language Support in *Japanese Language Support User's Guide.*

Introduction

The AIX Operating System and the C and FORTRAN programming languages are based on English language characters and symbols common to the United States of America. Character sets and date- and time- string handling for other languages and places are provided by AIX in a variety of formats. Conversion subroutines provide a means of translating between various character sets and date and time string formats.

AIX starts with the 7-bit ASCII character set, which represents 128 logical symbols. This character set represents digits, English language characters and punctuation, and a set of (nonprinting) control characters. Environment variables provide alternate time, date, and currency formats and select the characters to use for currency and other symbols. Pointers to tables provide characteristics used in character collation and comparison. Appropriate defaults are associated and can be selected by language.

AIX provides international character support in the form of an extended character set, which enables you to place various accent marks on characters and to supply characters and symbols used by many languages. The character set is extended to 2-byte characters by the **NLchar** data type. Conversion routines permit passing information between AIX systems that use these extended character sets and systems or terminals that do not use extended characters.

An ***extended character*** is defined as a non-ASCII character (upper code page P0, characters 128-255), or a character that is 2 bytes long (code page P1, for example).

Japanese Language Support Information

An extended character is defined as a non-ASCII character that consists of either 1 or 2 bytes.

Notes:

The following items can contain *only* ASCII characters when using Basic Networking Utilities:

a. System names
b. User names
c. File names.

Multihop transfers of files containing characters from the extended character set through non-AIX systems can have undetermined results.

A ***code page*** is an array of code points representing a character set. The P0 code page characters, described in *RT Keyboard Description and Character Reference*, are 1-byte characters that include the ASCII character set and other symbols and characters, including the control codes. The ASCII character set is extended by using four values (0x1F, 0x1E, 0x1D, and 0x1C) as single-shift (escape) bytes. Together with the next byte in the data stream, these single-shift bytes select a character from code page P1 or P2.

Additional code pages provide other characters and symbols to support languages or dialects of a site or user. (Japanese Language Support does not use code pages.)

Code points are conceptual characters that are either 1-byte or 2-byte objects. Code points represent both ASCII and extended characters.

A range of code points is established in ***collating sequence***, which sets up a lexicographical (alphabetic) order. A collating sequence also sets up equivalence classes. An ***equivalence class*** of code points is based on the natural groupings of characters for a language or dialect. (Japanese Language Support does not support equivalence classes.)

The **ctype** and **conv** groups of character conversion macros and subroutines are extended to permit code points to be identified and processed (see *AIX Operating System Technical Reference*). Macros and subroutines are defined to operate on the character type **NLchar** in a manner analogous to the macros and subroutines that handle **char** character type. Additional macros and subroutines provide translation between character types. Dynamically configurable tables are loaded into memory at run time to support the **NLchar** data type. (Dynamically configurable tables do not exist in Japanese Language Support.)

The **NLFILE** environment variable can be used to select conversion tables (**/usr/lib/nls/***file***.en**) that translate date, time, and currency information into an appropriate format for a selected nationality. A custom table can be prepared to support dialects, other languages, or the special needs of a site or an individual user. See **ctab** in *AIX Operating System Commands Reference* and **environment** in *AIX Operating System Technical Reference* for information on selecting formats and currency information.

Features

AIX provides these subroutine features for international character support:

- Character conversion and character collation.
- Regular expression support for non-ASCII characters. A code point is treated as one character, whether it is one or two bytes long. Metacharacters retain their traditional interpretation, but now refer to code points rather than bytes.
- Revision of character set, collating sequence, and equivalence classes of characters from an application by selecting a collating table input file using the **NLgetctab** subroutine. When a member of an equivalence class is used in a character range, all members of the class are treated as part of the range.
- Extension of all user-visible names by code points, permitting extended characters in names of files (including directory, device, and file-system names), and in user, group, and queue names.
- Support for input and output of code points using noninternational character support (ASCII) terminals. Characters entered at a keyboard are translated into code points

when they are input, then translated back into a mapped character set when they are output. Different terminal characteristics are supported.

- Conversion to and from ordinary ASCII text while preserving the essential characteristics of a file containing extended characters.
- Production of dialectic formats for dates and times. All AIX utilities can output time and date information in the selected configuration. The **at** utility accepts input dates in the currently configured format.
- Time zone support enabling the operating system automatically to begin and end daylight savings time as specified, using the **TZ** environment variable.
- Selection and placement of the currency symbol, and selection of the decimal point character and the thousands-divider character. An example of a U.S. default is `$9,999.99`.
- Conversion between code points and characters. The **NCdecode** and **NCencode** subroutines translate between **char** and **NLchar** type variables.
- Support for character strings. The **NCdecstr** and **NCencstr** subroutines convert a string of extended characters to ASCII, or a string of ASCII code points to extended characters.
- Retrieval from a collating table of information on the collating value of code points from the arrays created by the **ctab** command. The **NCcollate**, **NLstrcmp**, and **NCstrcmp** subroutines retrieve collation information. The **NCeqvmap** subroutine determines equivalence classes. (The **NCeqvmap** subroutine is not available when Japanese Language Support is installed.)
- Derivation of information on code points. The **NLchrlen**, **NCchrlen**, and **NLisNLcp** macros differentiate between characters and code points. The first two macros return the length of **chars** and **NLchars**. The third macro identifies code points and returns their length.
- Code point conversion without loss of character information. A code point can be converted to the single ASCII character it most resembles by the **NLflatchr** subroutine. (The **NLflatchr** subroutine is not available when Japanese Language Support is installed.) A code point can be converted to an ASCII escape sequence using the **NCesc** subroutine. The escape sequence can be reconverted to the original code point using the **NCunesc** subroutine.
- String manipulation support. The basic string manipulation subroutines can handle extended character strings. The **NCstring** and **NLstring** subroutine classes are provided to parallel the **string** subroutines.
- Code point extensions to I/O and in-memory format conversion subroutines. The standard conversion subroutines of the general **printf** and **scanf** types have been paralleled to handle extended character strings using the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines, which are provided for **NLchars**.

- Internal conversion between time formats and input and output time-format strings. The **NLtmtime** and **NLstrtime** subroutines convert a binary-format representation of time to or from a text-string representation of time, and the output is formatted according to conventions that the user can specify.
- Automatic initialization of international character support at AIX initialization. The **NLgetenv** subroutine reads the environment file to find an environment parameter for the collation table. If the environment does not define a collation table, **NLfile** is searched for a table assignment. A default table is used if the collation table is not found.

Japanese Language Support Information

In Japanese Language Support,the default collating sequence, based on encoded character values, is used.

Programming Language Support

AIX provides international character support features in C and FORTRAN languages.

Extensions to C libraries provide international character support for C programming. Macros and functions support character sets of a variety of languages by using the code point mechanism. Character constants are 1-byte objects. However, comments and character strings can have arbitrary contents and thus can contain international character support code points.

Note: The C programming libraries include an object called an **NLchar**, which is defined in **/usr/include/NLchar.h** as a short integer. An **NLchar** is a representation for a code point that can be readily manipulated in C. Two byte code points form 2-byte extended characters, and 1-byte code points comprise the 7-bit ASCII character set and 1-byte extended characters.

There is no provision to represent a 2-byte character as a single-quoted constant in a source file, but double-quoted strings are fully supported. Consider a construct like the following:

```
NLchar *xc;
.
.
.
if (*xc == 'œ'.) . . .
```

where œ is an extended character. In the case of an extended character, the **if** statement should instead read something like the following:

```
if (*xc == NCdechr("œ")) . . .
```

If a multibyte single-quoted constant is encountered, the AIX C compiler issues a warning message.

FORTRAN international character support is provided with string-compare functions, subroutine support for the Graphics Support Library, and other FORTRAN functions.

Regular expression support of extended characters is deactivated by defining the **regexp.h** header file instead of **NLregexp.h**. This lets you create applications for systems that do not support extended characters.

Kernel Modifications

The kernel is modified for terminal map support and to handle code points.

Terminal Maps

Terminal maps are provided to define a keyboard and display for international character support.

Additional maps are defined in files to support features of some terminals. The terminal type selection, which you can set with the **TERM** environment variable, does not always support the full range of characters that can be displayed to provide international character support. You can set the terminal map values from a file in the **/etc/ports** directory. Two port control parameters provide international character support (see **ports** file format in *AIX Operating System Technical Reference*). The **imap** parameter sets the terminal input map for the **getty** command, and **omap** sets the terminal output map.

You can also change the terminal map by using the **stty** command. This is useful when using a modem to log in. See "Character Display and Handling" on page 14-12.

Code Point Support

The kernel represents both **char** and **NLchar** data types internally as code points.

Each **NLchar** occupies 16 bits of storage. The high order bit of an **NLchar** is not used in representing the character. Library functions can be used to convert code points to and from **NLchar**s within an application.

Tables of mapped characters are provided in *RT Keyboard Description and Character Reference*. The ordinal ranges of the characters can be seen in the description of the **display symbols** subroutine in *AIX Operating System Technical Reference*.

System Representation Byte(s)		Binary NLchar Representation	Decimal Range
	0xxxxxxx	000xxxxxxx	0-127
	1xxxxxxx	001xxxxxxx	128-255
00011111	1xxxxxxx	010xxxxxxx	256-383

Figure 14-1 (Part 1 of 2). Code Points

System Representation Byte(s)	Binary NLchar Representation	Decimal Range
00011110 1xxxxxxx	011xxxxxxx	384-511
00011101 1xxxxxxx	100xxxxxxx	512-639
00011100 1xxxxxxx	101xxxxxxx	640-767

Figure 14-1 (Part 2 of 2). Code Points

A code point is removed from input with a single backspace character when **ICANON** is set. (See **terminfo** in *AIX Operating System Technical Reference.*) This causes a code point to be handled like a character. Otherwise, a single backspace character would only remove the second byte of a 2-byte code point. **ICANON** is set by default.

Note: Japanese Language Support does not support code points.

Environment Variables

Environment variables provide input and output support for preferences in formatting displayed information. Commands that display date or time information format input according to parameters read from the environment. Commands that accept a date or time string as command line input (**at**, **date**, and **touch**) parse their input according to the same environment parameters.

The following variables, which are described in "environment" in *AIX Operating System Technical Reference*, provide an international character support environment for a user.

NLFILE Names the default-value table file containing definitions of all other **NLenv** variables. U.S. English defaults are used if a value cannot be found in the environment specifications or in NLFILE. NLFILE table files, which have an **.en** suffix, reside in the directory **/usr/lib/nls**.

NLLANG Names the conventional language in use.

NLCTAB Selects the table file for character conversion and collating. The tables are accessed using a shared memory segment.

NLCURSYM Selects the conventional currency symbol and sets its placement.

NLDATE Describes the format for a "short" date. It is used for displaying and parsing dates.

NLLDATE Describes the format of a "long" date. It generally provides a date in the form of a string, with the month given as a word or word abbreviation rather than as a number. It is generally used for parsing dates.

NLLDAY Describes the format for "long" names of the days of the week. The form is not case-sensitive on input. For example, some utility might accept `mon` as input and output `Monday`, following the form of the variable parameter.

NLSDAY Describes the format for "short" days of the week (generally abbreviations).

NLLMONTH Describes the format for "long" names of months.

NLSMONTH Describes the format for "short" names of months.

NLTIME Describes the format of the time string.

NLTMISC Describes miscellaneous strings used for input of date and time to the **at** utility, with the time based on a 12-hour clock. The strings are translations of the U.S. English terms of the default strings.

NLTSTRS Describes strings of time-unit names used for input of date specifications to the **at** utility. These strings are translations of the U.S. English terms of the default strings.

NLTUNITS	Describes strings giving time-unit names used for input of date specifications to the **at** utility.
	Japanese Language Support Information
LANG	Names the conventional language in use.
NLSPATH	Describes the search path to be used when locating a message catalog.
NOSTR	Describes the allowed form for negative responses to system prompts.
YESSTR	Describes the allowed forms for affirmative responses to system prompts.

Character Display and Handling

You can use the **stty** command (described in *AIX Operating System Commands Reference*) to remap the input and output maps for improved international character support. (See "Kernel Modifications" on page 14-8.)

An application can handle **NLchar** data with a number of subroutines to provide international character support. With a few exceptions, these subroutines act as do their counterparts on **char** data type, and they are named like their counterparts with the prefixes **NC** and **NL**:

NC Used for routines that handle **NLchar** data.

NL Used for routines that handle byte strings that possibly contain extended characters.

Conversion between char and NLchar

Conversion between **char**s and **NLchar**s is performed by the **NCencode** and **NCdecode** subroutines, and the **NCenchr** and **NCdechr** macros.

The **NCencode** subroutine converts an **NLchar** to one or two **char**s and returns the number of **char**s produced. The **NCisNLchar** macro is provided to verify, where necessary, that the input is a valid **NLchar**.

The **NCdecode** subroutine translates one or two **char**s to an **NLchar** and returns the number of **char**s converted. The **NCdechr** macro works like **NCdecode** but returns the **NLchar**.

The **NCenc** and **NCdec** macros are defined as equivalents to **NCencode** and **NCdecode** for use where call overhead is to be avoided, and where parameters will not have side effects.

These conversion subroutines and macros return an invalid single-shift byte (a single-shift byte that is not followed by a byte with its high bit set), as they would any ASCII character. The application must handle an invalid byte as required.

Conversion between ASCII and NLchar

An **NLchar** can be converted to ASCII in either an ***information-preserving*** or an ***appearance-preserving*** conversion. The **NCesc** macro converts a single **NLchar** to an ASCII escape sequence that uniquely identifies a code point, preserving the information in a string of the characters of this form:

\<*cc*>

For example, the character œ converts to the ASCII string \<oe>. (The **display symbols** function, described in *AIX Operating System Technical Reference*, lists these sequences.) The *cc* is a mnemonic character or string that corresponds to the **NLchar**, while the two leading characters and the trailing characters are literal. The **NCesc** macro returns the number of bytes it stored. See the description of the **display symbols** subroutine in *AIX Operating System Technical Reference* or examine the table in **/usr/pub/charset.ibm** for a listing of the characters and their mnemonic conversion strings.

The **NCunesc** macro converts an ASCII escape sequence to the unique extended character it represents.

The **NCflatchr** subroutine simply converts the **NLchar** into the ASCII character it most resembles, using a fixed table.

Japanese Language Support Information

An **NLchar** can be converted to ASCII *only* in an information-preserving conversion. The **NCflatchr** subroutine is not active in Japanese Language Support.

Character Information Macros

Three macros return information about a single character represented as an **NLchar** or as a part of a **char** string.

NLchrlen Returns the length in bytes (one or two) of the character pointed to by a pointer of the type **char**.

NCchrlen Returns the length that an **NLchar** would be if it were represented as bytes.

NLisNLcp Returns the length of a code point pointed to by a **char** pointer; if the character pointed to is not a valid code point, it returns 0 (zero).

The NLyesno Subroutine

The **NLyesno** subroutine provides a language-independent means of determining which forms the system allows for affirmative and negative responses to prompts.

The **NLyesno** routine performs this function by comparing the string *s* with the substrings given in the values of the environment variables **YESSTR** and **NOSTR**. An affirmative response to a system prompt is acceptable if the *s* string matches an element of **YESSTR**. A negative response is acceptable if *s* matches an element in **NOSTR**.

Note that a null response is indicated in **YESSTR** or **NOSTR** by a leading or trailing **:** (colon), or by **::** (two adjacent colons). The string *s* is considered a null response if its first element is either \0 or \n.

String Handling

String functions are provided to handle code points. They correspond to the traditional string functions, but have an **NL** prefix. These subroutines are functionally identical to the corresponding **string** subroutines: **NLstrcat**, **NLstrncat**, **NLstrcpy**, **NLstrncpy**, and **NLstrlen**.

The **NLstrchr** and **NLstrrchr** functions can substitute for **strchr** and **strrchr**, but not the reverse. They can accept **NLchar**s for their second parameter.

The remaining **NLstring** functions are not compatible with their **string** counterparts: **NLstrcmp**, **NLstrncmp**, **NLstrpbrk**, **NLstrspn**, **NLstrcspn**, and **NLstrtok**.

Another set of string subroutines are provided to operate on **NLchar** strings; they have **NC** prefixes. These functions correspond to their **string** equivalents, except that they require **NLchar**s instead of **char**s. However, the functions **NCstrpbrk**, **NCstrspn**, **NCstrcspn**, and **NCstrtok** will accept a string of type **char** as their second parameter.

The **NCstrlen** function works like the **strlen** function except that each code point counts as a unit in length calculations.

String-handling functions corresponding to the **printf** and **scanf** functions are provided, with an **NL** prefix. These are extended with new kinds of formats.

Intersystem Compatibility

AIX retains compatibility with noninternational character support environments to support work stations, intersystem mail, networking, and program development for systems that do not provide international character support. The same mechanisms that provide translation between environments with and without international character support also provide translation between systems that use different international character support environments.

It is not always possible to use terminal mapping during the login procedure; a user can access a remote port with one of several types of work station at any time. Therefore, an ASCII-character synonym is automatically defined for a login name containing extended characters to allow logging in from a work station that does not support extended characters.

The synonym is mapped consistently using the **NCflatchr** macro at the time of login, so that ASCII synonyms do not have to be explicitly maintained in the **/etc/passwd** file. The **adduser** utility checks both the login name containing extended characters and its ASCII synonym against all existing full IDs and their ASCII synonyms before permitting the login name to be added to the system.

AIX can communicate with other systems through communications protocols such as the Basic Networking Utilities (BNU) or **SNA**. Site names must conform to the requirements of the host system communication protocol (currently **SNA** and BNU) and must be composed of ASCII characters. Therefore, site names are limited to ASCII characters or some subset of ASCII characters.

Japanese Language Support Information

An extended character is defined as a non-ASCII character that consists of either 1 or 2 bytes.

Notes:

1. The following items can contain *only* ASCII characters when using Basic Networking Utilities:

 a. System names
 b. User names
 c. File names.

 Multihop transfers of files containing characters from the extended character set through non-AIX systems can have undetermined results.

2. The **NLflatchr** subroutine is not active in Japanese Language Support.

The sender can translate all user names or alias names to ASCII characters using the **dd** command, with the receiver reading the ASCII characters or, if conversion was made to

escape sequences to preserve extended character meanings, retranslating the characters to extended characters. The translation would be transparent to AIX hosts that provide international character support.

The **dd** utility provides translation of files containing extended characters when copying files between hosts. You can use translation subroutines and macros to translate characters and strings as necessary from an application.

A discussion of changes required to support BNU communications is provided below to illustrate the scope of support needed for communications between systems.

Basic Networking Utilities

No translation occurs to or from extended characters when the BNU **uucp** command is used to transfer files locally. To transfer a file from one remote system to another, the **uucp** command adds a line to a command (work) file that causes the remote system to generate a BNU command.

The additional command line is ignored by systems without international character support. **uucp** converts the extended characters of a file name that appears on a work file command line to ASCII characters. Escape sequences represent the extended characters. A receiving system with international character support translates the file name to extended characters as needed, using **getpwnam**, while a system without international character support uses the 1-byte characters without translation. Date and time strings are output in the format established by environment variables for international character support.

Note: Transmitting code points between systems that are not using the same character set can produce unpredictable results.

Japanese Language Support Information

An extended character is defined as a non-ASCII character that consists of either one or two bytes.

The following items can contain *only* ASCII characters when using Basic Networking Utilities:

1. System names
2. User names
3. File names.

Multihop transfers of files containing characters from the extended characters set through non-AIX systems can have undetermined results.

To send a file, **uucp** adds a line to the work file that contains ASCII path names of the source and destination files, the user's login name, the name of the data file in the spool

directory, and the user ID to notify on the remote system. It copies the source file into a data file in the spool directory.

To receive a file, **uucp** adds a line to the work file that causes the remote machine to execute a **uucp** command. The source file can contain metacharacters, which refer to code points that **uucp** translates to actual file names composed of 1-byte characters.

To be compatible with **uucp** requirements, many associated commands are revised to provide international character support:

uucico The secondary side recognizes a command identification character to distinguish between actual file names and file names converted to 1-byte characters.

The **L.sys** file that **uucico** uses to establish "days" does not use standard abbreviations for days and uses some terms for which there is no equivalent in international character support. The manner in which days are specified is unchanged, although time and date are printed in a format for international character support.

uuclean Uses **qetpwnam** to derive the extended character name of the user to notify that files have been removed.

uulog Uses **NCflatchr** to provide an ASCII user name, supporting calls with a **-u** flag. **uulog** uses the **NLDATE** environment variable to set the **LOGFILE** date format. (When Japanese Language Support is installed on your system, **NCflatchr** is not supported.)

uustat Uses **NCflatchr** to provide an ASCII user name, supporting calls with a **-u** flag. (When Japanese Language Support is installed on your system, **NCflatchr** is not supported.)

uusub Checks to assure there are no extended characters in traffic between systems.

uux The secondary side recognizes a new command identification character to distinguish between actual file names and file names that have been converted to ASCII strings.

A **uux** command, generated by a **uucp** command with the **-e** flag (to cause **uucp** to be executed by the remote system), treats command line file names as does **uucp**.

uuxqt Recognizes the command identification character to distinguish between actual file names and file names converted to ASCII strings.

The preceeding commands print time and date formatted for international character support.

The **uuname**, **uuto** and **uupick** utilities are unchanged. Note that system names cannot contain non-ASCII characters.

The international character support for communications between systems is described further in *AIX Operating System Communications Guide.*

Limits to Support

Some subroutines that use or control the system, or pass characters to or from it, do not handle **NLchars**. Many library subroutines are unchanged, and strings passed to and from them must be explicitly converted using international character support macros.

The **getopt** subroutine does not allow extended code points as option letters, to retain compatibility with systems without international character support or supporting a different language or dialect.

File and directory names have a size limit of 14 bytes. Because code points can represent either one or two bytes of storage, the number of significant code points in a file name or directory name can be limited to as few as seven. File and directory names longer than 14 bytes are truncated to a maximum of 14 significant bytes; if the fourteenth byte is the first byte of a 2-byte code point, the name is truncated to 13 bytes.

Collation

You can use the **ctab** command to create a new table defining the collating sequence. You can also use **ctab** to create a new table defining the case conversion of characters. (Japanese Language Support does not support character tables.)

Note: The input and output files are stored in the conventional directory **/usr/lib/nls**. Files exist to support various user environments, and you can always create new files with names that reflect a specific environment. In an environment in which users work with files that include British characters, symbols, and lexicographical collating sequences, for example, file names might include **uk** for "United Kingdom."

The **ctab** command enables you to use a defined table file for a language without changes. You can also modify an existing table, or build a customized table. The **NLgetctab** subroutine gets an output file at initialization and loads it into memory, while the **NLgetenv** subroutine can select a different table file from an application.

The table input file provides the collating sequence for each character in the table (each **NLchar**).

The table input file also provides the following information:

- For characters having case, the corresponding uppercase or lowercase version of the given character.
- The characters in an equivalence class (used for regular expression range processing). All the characters in this class are counted in the character range, along with this character, whenever this character is named as an end point of a character range.

Additional information about setting up collation tables is also contained in "Overview of International Character Support" in *Managing the AIX Operating System.*

For information about setting **ctab** attributes, see the description of the **ctab** command in *AIX Operating System Commands Reference.*

Files and Directories

Some files are modified and others are added to provide international character support. New directories contain groups of files for international character support.

/usr/lib/nls/*	Contains character collation tables and *ctable*.**en** environment files. Input tables are of the form *ctable*.**ctab**, and output tables are binary arrays of the form *ctable*. When AIX is installed on your system, the default collation table, used when no table is set by an environment variable, is **/etc/nls/ctab/default**.
/etc/profile	Provides environment variables for all users (A shell script).
/etc/environment	Provides system environment variables (A shell script).
/$HOME/.profile	Provides user-specific variables.
/usr/include	Holds header files that provide international character support. These include the following: **/usr/include/NLregexp.h** Provides regular expression support for the extended character set. **/usr/include/NLctype.h** Provides support for character type conversion. **/usr/include/NLchar.h** Provides support for **NLchar** character type.
/usr/include/sys	Some header files in this directory are modified to provide support for terminal mapping. This includes the following: • **termio.h** • **ttmap.h** • **tty.h**
/etc/nls/termmap/*	Contains terminal maps.
/etc/nls/ctab/*	Contains NLS conversion table for the language you selected when installing your system.

Chapter 15. Using Remote Procedure Call (RPC)

CONTENTS

About This Chapter 15-2
Overview of RPC 15-3
The RPC Communication Paradigm 15-3
Data Transports and Semantics 15-4
Binding and Rendevous Independence 15-4
Message Authentication 15-4
The RPC Protocol 15-4
Remote Procedure Call Language (RPCL) 15-6
Defining Arbitrary Data Types with eXternal Data Representation (XDR) 15-12
RPC Programming Levels 15-18
Using the Highest Level 15-18
Using Intermediate-Level RPC 15-20
Using Low-Level RPC 15-22
Authenticating Remote Procedure Calls 15-30
Authenticating the Client 15-30
Authenticating the Server 15-31
Special Topics 15-35
The select Procedure on the Server Side 15-35
Broadcast RPC 15-37
Using Batching 15-39
Using the inetd Daemon to Start a Server 15-43
Example Applications 15-45
Example of Using Version Numbers 15-45
Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP) 15-47
Example of Using Callback Procedures 15-53

About This Chapter

This chapter describes how to write network applications using the Remote Procedure Call (RPC)[1] specification. Remote procedure calls are used to communicate between work stations in a local area network. This allows users to share files or access additional disk space without leaving their work stations.

The chapter begins with an overview of the RPC specification, and the software components that support it: Remote Procedure Call Language (RPCL)and eXternal Data Representation (XDR)[2]. Next in the chapter is a discussion of the three levels of the RPC interface and how to authenticate the remote procedure calls. The chapter concludes with sections that contain examples of specific network applications of the RPC interface.

For information on specific RPC routines, see the Remote Procedure Call and RPC Service Routines sections in *AIX Operating System Technical Reference.*

1 The Remote Procedure Call interface was developed by Sun Microsystems, Inc. RPC is a trademark of Sun Microsystems, Inc.

2 Remote Procedure Call Language and eXternal Data Representation were developed by Sun Microsystems, Inc.

Overview of RPC

RPC is a remote procedure call specification that provides a procedure-oriented interface to remote services. RPC is used in networks to provide programs that enable communication between machines. For example, a network file service can be composed of programs that deal with high level applications such as file access control, and programs that deal with low level applications such as **read** or **write**. The programs are accessible through a machine designated as a network server. A client of the network file service can call the procedures associated with the programs on behalf of a user logged in to the client machine.

A ***client*** is a computer or process that accesses the services or resources of another process or computer on the network. A ***server*** is a computer that provides services and resources, as well as implements network services. Each network service is a collection of remote programs. A remote program implements remote procedures. The procedures, along with their parameters and results, are documented in the specific program's protocol specification. A server can support more than one version of a remote program in order to be compatible with changing protocols.

In RPC, each server supplies a program that is a set of procedures. The combination of a host address, a program number and a procedure number specifies one remote service procedure.

The RPC Communication Paradigm

Programs that communicate over a network need a paradigm for communication. The RPC paradigm is based on the remote procedure call model, which is similar to the local procedure call model. A local procedure call involves the caller placing arguments to a procedure in a defined location, such as a result register, and transfering control to the procedure. The caller eventually gains back control, and extracts the results of the procedure from the defined location before continuing execution.

The remote procedure call is similar, except that one thread of control winds through two processes: a caller process and a server process. That is, the caller process sends a call message to the server process and waits (or blocks) for a reply message. The call message contains information that includes the parameters of the procedure. The reply message contains information that includes the results of the procedure. When the caller receives the reply message, it extracts the results of the procedure and resumes execution.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and awaits the next call message.

Only one of the two processes is active at any given time. That is, the RPC protocol does not explicitly support multithreading of caller or server processes.

Data Transports and Semantics

RPC deals with the specification and interpretation of messages, not with the method used to pass messages from one process to the other. It does not depend on services provided by specific transport protocols. Although specific semantics, or meanings, are not attached to remote procedures or their execution, certain semantics can be inferred from the protocol of the underlying data transport that is used.

For example, passing RPC messages with the UDP/IP data transport is unreliable. If the caller retransmits RPC call messages after short timeouts, the only thing it can infer from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). In contrast, passing RPC messages with TCP/IP is reliable. No reply message means the remote procedure was executed one time at most, and a reply message means that the remote procedure was executed exactly once.

Binding and Rendevous Independence

RPC does not bind a client to a service as part of its protocol. This required function is left up to a higher level software. However, the network software can use RPC to accomplish the tasks involved with binding clients to services.

Message Authentication

The RPC protocol provides the fields required for a client to identify itself to a service, and for a service to identify itself to the client. The contents of RPC authentication parameters for these fields are determined by the type, sometimes called flavor, of the authentication used by the server and the client. A server can support multiple types of authentication at one time.

You can build additional security and access controls on top of the message authentication.

The RPC Protocol

RPC is primarily a tool for calling remote procedures. By providing a unique specification for calling the remote procedures, RPC can match a reply message to each request (or call) message.

Each RPC call message contains the following unsigned fields to uniquely identify the procedure to be called:

- Remote program number
- Remote program version number
- Remote procedure number.

Assigning Program Numbers to Protocols

Program numbers are assigned in groups of 0x20000000 (536870912) as shown in Figure 15-1:

Program Number	How Assigned	Use
0 - 1fffffff	Defined by system authority	System authority (the product licensor) administers this first group of numbers. This group should be identical for all system customers.
20000000 - 3fffffff	Defined by user	Use this group for applications you develop and for debugging new programs.
40000000 - 5fffffff	Transient	Use the third group for applications that generate program numbers dynamically.

Figure 15-1. How to Assign Program Numbers

Assigning Version Numbers to Programs

As programs evolve into more stable and mature protocols, version numbers are assigned. The first implementation of a remote program is usually designated as `version number 1` (or a similar form).

The version number identifies which version of the protocol the caller is using. Version numbers make it possible to use old and new protocols through the same server.

Assigning Procedure Numbers to Programs

The procedure numbers are documented in each program's protocol specification. For example, a file service protocol's specification can list the **read** procedure as procedure number 5 and **write** as procedure number 12.

Remote Procedure Call Language (RPCL)

RPC uses the Remote Procedure Call Language (RPCL) as the input language to its protocols and routines. RPCL specifies the data types used by RPC and generates the XDR routines that standardize their representation.

In order to implement the service protocols and routines, the RPCL input is compiled into the corresponding C language code using the **rpcgen** command. The C code is compiled by converting the RPCL definitions to C language definitions and placing them in a header file. The **rpcgen** command also compiles the corresponding XDR routine that serializes the protocol. The RPCL input can contain comments and preprocessor directives, but **rpcgen** ignores comments in the text and copies the directives to the output header file without interpreting them. For more information on the **rpcgen** command, see *AIX Operating System Commands Reference.*

The following section contains brief definitions of the RPCL syntax intended to help you understand RPCL. These definitions are not exact statements of the language protocols.

Primitive Data Types

RPCL uses the following primitive data types:

char	A single character value. A **char** value can accept a sign (positive or negative).
unsigned_char	A single character value on which a sign extension cannot occur.
int	An integer or numerical representation.
unsigned_int	An integer value that is not negative and has the same number of bits as an **int**.
long	An integer value that has the same or a larger number of bits as an **int**.
unsigned_long	An integer value that is not negative and has the same or a larger number of bits as an **unsigned_int**.
short	An integer value that has the same or a smaller number of bits as an **int**.
unsigned_short	An integer value that is not negative and has the same or a smaller number of bits as an **unsigned_int**.
float	Single floating-point number value. A single floating-point number is a number that contains an exponent value. It can also contain a fraction.
double	A floating-point number value that has the same or a larger number of bits as a **float**.
void	No data structure. The **void** declarations can appear only inside **union** and **program** definitions.

bool Boolean value. The **rpcgen** command converts **bool** declarations to **int** declarations in an output header file.

Note: The prefix **unsigned** can be shortened to **u_** so the data types can be abbreviated as follows: **u_char**, **u_int**, **u_long**, and **u_short**.

Defining Other Data Types

When you use RPCL data declarations, you cannot declare multidimensional arrays. You cannot point to pointers inline, but you can declare them with **typedef** statements. In addition, you can declare opaque data and strings as vectors. The vector declaration for opaque data is as follows:

`opaque` *objectident* [*size*]

A vector resulting from an opaque data declaration is compiled in the output header file as a character array of the size, in bytes, specified by the *size* parameter. Do not confuse this array declaration with the declaration of character sizes of XDR characters because the XDR characters are defined as 32-bits. The vector declaration for string data is as follows:

`string` *objectident* [*maxsize*]

The *maxsize* parameter specifies the maximum size, in bytes, of the vector representing the string. If you do not specify *maxsize,* there is no limit to the maximum length of the string. String declarations are compiled into a character string pointer that points to the object. The character string appears in the following form:

`string` **object-ident*

Structure and Type Definitions

The only way to generate an XDR routine is to define a data type. For example, if you define a data type named `zetype`, an XDR routine called `xdr_zetype` is generated to serialize it. You cannot nest data type definitions. Nesting them can cause the **rpcgen** command to fail when it tries to compile the C code for the definition.

Declaring Arbitrary Types with typedef

The **typedef** declarations are similar to the **typedefs** in the C language. They take the following simple declaration form:

`typedef` *declaration* ;

The *declaration* parameter contains the *typename* and *objectident* parts from the simple declaration form. The *typename* parameter specifies the name of the data type from which

the new data type is derived. The *objectident* parameter specifies the name of the new data type.

Declaring Enumerations

The **enumeration-def** declarations are similar to their C language counterparts. They take the following form:

```
enum enumident {
    enumlist
};

enumlist:
    enumsymbol-ident [ = assignment ]
    enumsymbol-ident [ = assignment ], enumlist
```

The *assignment* parameter is an integer or a symbolic constant. If there is no explicit *assignment*, the implicit *assignment* is the value of the previous enumeration plus 1. If not explicitly assigned, the first enumeration receives the value 0.

Declaring Structures

The **structure-def** declarations are similar to their C language counterparts. They take the following form:

```
struct struct-ident    {
    declaration-list
};

declaration-list:
    declaration    ;
    declaration    ; declaration-list
```

Declaring Variable Length Arrays

The **variable-length-array-def** declarations are unique to RPCL. They take the following form:

```
array array-ident {
    unsigned length-identifer ;
    vector-declaration ;
};
```

A variable length array is similar to a structure. The following example shows a variable length array definition:

```
array mp_int {
    unsigned len;
    short val[MAX_MP_LENGTH];
};
```

compiled into the following structure definition:

```
struct mp_int {
    unsigned len;
    short *val;
};
typedef struct mp_int mp_int;
```

Declaring Discriminated Unions

The *discriminated-union-def* declarations differ from the standard C union. The discriminated unions are unions that have a value that is added as an arm of the union to specify a particular object. RPCL discriminated union definitions take the following form:

```
union union-ident switch (discriminant-declaration) {
case-list
[ default : declaration ; ]
};

case-list:
    case case-ident : declaration ;
    case case-ident : declaration ; case-list
discriminant-declaration:
    declaration
```

The union definition appears as a cross between a C union and a C switch. The following is an example of a union definition:

```
union net_object switch (net_kind kind) {
case MACHINE:
    struct sockaddr_in sin;
case USER:
    int uid;
default:
    string whatisit;
};
```

compiled into the following:

```
struct net_object {
    net_kind kind;
    union {
        struct sockaddr_in sin;
        int uid;
        char *whatisit;
    } net_object;
};
typedef struct net_object net_object;
```

Note that the output structure's union component name is the same as the data type name.

Declaring Program Definitions

The **program-def** declarations do not define data type. They define information used by the client programs to reference remote procedures. The **program-def**s take the following form:

```
program program-ident {
        version-list
} = program-number ;

version-list:
        version
        version version-list

version:
        version version-ident {
                procedure-list
        } = version-number ;

procedure-list:
        procedure-declaration
        procedure-declaration procedure-list

procedure-declaration:
        type-name procedure-ident (type-name) = procedure-number;
```

Client programs use program definitions from these declarations to reference the remote procedures associated with a user commands execution. When a **program-def** is compiled, it becomes a **#define** statement that identifies a remote procedure. When the server receives the local procedure, it matches the function of the procedure to an existing C function of the same name. Although the C function has the same name, it appears in lowercase letters and can be followed by a version number.

Note: XDR recursively frees the argument after getting the results from your local procedure, so be sure to copy from the argument any data you need between calls.

The following example illustrates the use of the **program-def** declarations. Suppose you wanted to create a server that can get or set the date. You can use **program-def** declarations to identify the remote procedures the program needs to access. The **program-def** declaration for this example could appear as follows:

```
program DATE_PROG {
    version DATE_VERS {
        date DATE_GET(timezone) = 1;
        void DATE_SET(date) = 2;
    } = 1;
} = 100;
```

This compiles in the header file as the following:

```
#define DATE_PROG 100
#define DATE_VERS 1
#define DATE_GET 1
#define DATE_SET 2
```

In the example, the local procedure defined by **#define DATE_GET** has the following form:

```
date *
date_get_1(tz)
    timezone *tz;
{
    static date d;

    return(&d);
}
```

Defining Arbitrary Data Types with eXternal Data Representation (XDR)

RPC handles arbitrary data structures in the remote procedure call message data by converting the structures to a network standard called eXternal Data Representation (XDR) before sending them through the system. The process of converting data from a machine's representation to the XDR format is called ***serializing***. As data is serialized, the size of each type is set to allow the data to be shared over the network independent of machine type or structure alignment algorithm. XDR serializing defines data externally in memory, allowing other functions using the same data to access the definitions.

The reverse process of serializing is called ***deserializing***. This converts the message data back to the original machine representation. XDR converts data quantities in multiples of 4 bytes when deserializing.

XDR routines are not direction-dependent. The same routines can be called to serialize and deserialize data types.

XDR routines return a non-zero number (or the value for `true` in the C language) upon successful completion. They return zero if the serialization or deserialization is not successful.

Data type parameters in RPC calls can be supplied from XDR's built-in routines, or from routines that you create. XDR contains the following set of built-in routines for the primitive data types:

- **xdr_int()**
- **xdr_u_int()**
- **xdr_long()**
- **xdr_u_long()**
- **xdr_short()**
- **xdr_u_short()**
- **xdr_bool()**
- **xdr_enum()**
- **xdr_string()**

Note: For specific information about these routines, see *AIX Operating System Technical Reference.*

Users can create their own XDR routine to serialize a data type that they have defined. For example, you can define a data type called `simple` with a structure declaration as shown in the following:

```
struct simple {
        int a;
        short b;
} simple;
```

In the `simple` structure, the parameters have integer and short integer values.

The corresponding XDR structure could be created as follows:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
        XDR *xdrsp;
        struct simple *simplep;
{
        if (!xdr_int(xdrsp, &simplep->a))
                return (0);
        if (!xdr_short(xdrsp, &simplep->b))
                return (0);
        return (1);
 }
```

Next, you can use the `simple` structure to call a remote procedure as shown in the following:

```
callrpc (hostname, prognum, versnum, procnum,
    xdr_simple, &simple ...);
```

Note: Programming with the **callrpc** routine is discussed in a later section of this book. For detailed information on **callrpc** and its parameters, see *AIX Operating System Technical Reference.*

In addition to the primitive data type routines, XDR also includes the routines for the following:

xdr_array()	Arrays of arbitrary elements of fixed length.
xdr_bytes()	Variable arrays of character bytes.
xdr_reference()	Pointers to other structures within structures.
xdr_union()	Discriminated unions which are comprised of a C language union and an enumerated value that represents an arm of the union.

Note: For specific information on these routines, see *AIX Operating System Technical Reference.*

These routines cover the constructed data types, which are more complex than the primitive data types.

XDR does not provide a routine for variable length arrays. To send a variable array of integers, the following structure can be defined:

```
struct varintarr {
        int *data;
        int arrlnth;
} arr;
```

The **xdr_varintarr()** routine is defined as follows:

```
xdr_varintarr (xdrsp, arrp)
        XDR *xdrsp;
        struct varintarr *arrp;
{
        xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, MAXLEN,
                sizeof(int), xdr_int);
}
```

The corresponding XDR routine takes as parameters:

- The XDR stream handle. The XDR stream handle identifies the XDR stream where data objects are serialized to XDR format and deserialized back into machine representation.
- Pointer to the data in the array. This is also where the data representation is placed for future use.
- Pointer to the size of the array.
- Maximum length of the array allowed.
- Size of each array element.
- The XDR routine that handles each array element.

With the variable array of integers defined, the following remote procedure call can be made:

```
callrpc (hostname, prognum, versnum, procnum,
    xdr_varintarr, &arr ...);
```

Note: Programming with **callrpc** is discussed in a later section of this book. For detailed information on **callrpc** and its parameters, see *AIX Operating System Technical Reference.* See the XDR library section of *AIX Operating System Technical Reference* for information on XDR routines.

Allocating Memory With eXternal Data Representation (XDR)

In addition to their input and output functions, XDR routines allocate memory. For this reason, the second parameter of XDR routines contains a pointer to the object instead of the object itself. In the case of the constructed data types, the sizes of objects are specified and placed in structures that are pointed to by the second parameter.

Allocating and freeing memory is usually not a concern unless a routine specifically directs it. However, a simplified example of memory allocation is shown below. For more information, see the XDR section in *AIX Operating System Technical Reference.*

In the following example, the **xdr_chararr1** routine deals with a fixed array of bytes with the length SIZE:

```
xdr_chararr1(xdrsp, chararr)
        XDR *xdrsp;
        char chararr[];
{

        char *p;
        int len;

        p = chararr;
        len = SIZE;
        return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

It can be called from a server using the following:

```
char chararr[SIZE];
svc_getargs(transp, xdr_chararr1, chararr);
```

where **chararr** has already allocated space.

If you want XDR to allocate the memory space, the routine can be written as follows:

```
xdr_chararr2 (xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
        return (xdr_bytes(xdrsp, charrarrp, &len, SIZE));
}
```

The RPC call can look like the following:

```
char *arrptr;
arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
svc_freeargs(transp, xdr_chararr2, &arrptr);
```

The **svc_freeargs** routine frees the character array.

For more information about the XDR routines, see the XDR library section of *AIX Operating System Technical Reference*.

RPC Programming Levels

The RPC interface is divided into the following three programming levels:

- At the ***highest level***, you can call RPC library routines directly in a program. The RPC library routines take care of making the remote procedure calls they need to function.
- At the ***middle level***, you can directly use the **registerrpc** and **callrpc** routines to make and execute remote procedure calls. Use this level for most common RPC applications.
- At the ***lowest level***, you can work directly with ***sockets*** that transmit the RPC messages. In AIX, ***sockets*** are the mapping of port numbers to Internet addresses in order to create a unique identifier for a host machine. Use this level for more direct control, such as changing the defaults of RPC routines and manipulating the sockets.

Using the Highest Level

When you use the highest level of RPC, you call RPC library routines to make the procedure calls in your programs. The library routines access RPC service procedures needed to make and execute the required remote procedures associated with a remote program. RPC is transparent to the programmer at this level because it resembles programming without the need to know RPC.

When you use a library routine in a program, the program must be compiled using the **rpcsvc** library file. Compile the program with the following command:

```
cc program.c -lrpcsvc
```

The following table contains the RPC library routines available for inclusion in your programs.

Routine	Description
getrpcport()	Gets RPC port number servicing the requested procedure's function.
rnusers()	Returns number of users on remote machine.
rusers()	Returns information about users on remote machine.
havedisk()	Determines if remote machine has disk.

Routine	Description
rstat()	Gets performance data from remote kernel.
rwall()	Writes to specified remote machines.
getmaster()	Gets name of YP master server if Yellow Pages is being used to administer the system.
yppasswd()	Updates user password in Yellow Pages databases if Yellow Pages is being used to administer the system.

Figure 15-2 shows how to use the **rnusers** library routine to determine the number of users logged in to a remote machine.

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned num;

    if (argc < 2) {
       fprintf(stderr, "usage: rnusers hostname\n");
       exit(1);
    }
    if ((num = rnusers(argv[1])) < 0) {
       fprintf(stderr, "error: rnusers\n");
       exit(-1);
    }
    printf("%d users on %s\n", num, argv[1]);
    exit(0);
}
```

Figure 15-2. Using the rnusers Library Routine in a Program

Using Intermediate-Level RPC

When you use intermediate-level RPC, you can work directly with the **callrpc** and **registerrpc** routines to make and execute remote procedure calls. The **callrpc** routine calls a remote procedure. The **registerrpc** routine matches the remote procedure number to its corresponding C language procedure.

Access to a remote procedure is through the program number, version number, and procedure number. The ***program number*** defines a group of related remote procedures. Each program has a ***version number*** to allow minor changes to a program, such as adding a new related procedure, without assigning a new program number. Each of the program's related procedures has a ***procedure number*** that uniquely identifies it in the group. The program, version and procedure number related to each remote service procedure is documented by RPC. You must look up the information to call a remote service procedure.

Using the callrpc Routine

Eight parameters are used to call a remote procedure with the **callrpc** routine. Besides the program, version and procedure numbers, the name of the remote machine on which the procedure resides must be identified. The next two parameters contain the arguments to the remote procedure call.

There are two more parameters that identify the return value of the call. One return parameter contains the return value itself. If the **callrpc** routine succeeds, it returns the value 0. If the routine does not succeed, it returns an integer value that represents a client condition. The meaning of each return code is defined in the **<rpc/clnt.h>** header file. The other return parameter points to the location of the return value.

Since internal machine data types can be represented differently on machines, you must supply the data type of the RPC argument as well as a pointer to the argument itself.

For example, the **RUSERSPROC_NUM** function takes no arguments, resides on a network server, and returns an unsigned long data type value. When you use the **callrpc** routine to call the **RUSERSPROC_NUM** function, **callrpc** uses the **xdr_u_long** routine as its first return parameter. The second return parameter of **callrpc** points to a host machine's pointer that indicates where the return value associated with **unsigned long** is placed. Since the **RUSERSPROC_NUM** function takes no arguments, the **xdr_void** routine is the the argument parameter of **callrpc**.

The **callrpc** routine tries several times to deliver the call message. If it repeatedly gets no reply, **callrpc** returns an error code transported by the ***User Datagram Protocol*** (***UDP***). The **callrpc** routine contains defaults which determine both the number of times **callrpc** tries to deliver a message and the use of UDP as the return value's transport protocol. See the **callrpc** routine in the RPC section of *AIX Operating System Technical Reference* for information on changing these values.

Figure 15-3 on page 15-21 shows how to use the **callrpc** routine in a program to determine the number of users logged in to a remote machine.

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
    int argc;
    char **argv;
{
    unsigned long nusers;

    if (argc < 2) {
       fprintf(stderr, "usage: nusers hostname\n");
       exit(-1);
    }
    if (callrpc(argv[1],
      RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
      xdr_void, 0, xdr_u_long, &nusers) != 0) {
        fprintf(stderr, "error: callrpc\n");
        exit(1);
    }
    printf("%d users on %s\n", nusers, argv[1]);
    exit(0) ;
}
```

Figure 15-3. Using callrpc to Determine Number of Remote Users

Using the registerrpc Routine

When network servers are booted, they register the RPC procedures they will handle with the **portmap**., and go into an infinite loop waiting to service remote procedure call requests from clients. The RPC procedures are registered by the servers using the **registerrpc** routine. The routine uses six parameters to register an RPC call. The first three parameters register the remote procedure's program, version, and procedure number. The fourth parameter registers the name of the C language procedure required to implement it. The last two parameters identify the data types of the procedure's input and output. (See the Remote Procedure Call section in *AIX Operating System Technical Reference* for information on the **registerrpc** routine.)

The **registerrpc** routine uses the User Datagram Protocol (UDP/IP) as its data transport when it registers an RPC call with a server.

Figure 15-4 shows how to use the **registerrpc** routine in a program to register RPC calls with a server.

```
#include <stdio.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
    registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
       nuser, xdr_void, xdr_u_long);
    svc_run();      /* never returns */
    fprintf(stderr, "Error: svc_run returned!\n");
    exit(1);
}
```

Figure 15-4. Using registerrpc to Register RPC Calls with a Server

Using Low-Level RPC

You can change the default values set for RPC routines and manipulate the sockets that transmit remote procedure calls by using the lower level of the RPC library. You should be familiar with sockets and their system calls to program with low-level RPC routines. See *Interface Program for use with TCP/IP* and *AIX Operating System Technical Reference* for information about sockets.

You can use the lowest level of the RPC library to do the following:

- To change to the Transmission Control Protocol/Internet Protocol (TCP/IP) data transport protocol from the default User Datagram Protocol/Internet Protocol (UDP/IP). UDP/IP transports packets of data that contain 8 kilobytes of data or less. TCP/IP transports the data in long streams of data.
- To allocate and free memory while serializing (or deserializing) message data with XDR (eXternal Data Representation) routines. You cannot free memory in the higher programming levels of RPC.

- To perform authentication on a client or server by supplying or verifying credentials.

Using the svc_register Routine

Calling the **svc_register** routine illustrates how low level RPC works. Figure 15-5 shows how to write a low-level RPC network application for determining the number of remote users logged in to a server by calling the **svc_register** routine.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

main()
{

    SVCXPRT *transp;
    int nuser();

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL){
       fprintf(stderr, "can't create an RPC server\n");
       exit(1);
    }
    pmap_unset(RUSERSPROG, RUSERSVERS);
    if (!svc_register(transp, RUSERSPROG, RUSERSVERS,
            nuser, IPPROTO_UDP)) {
       fprintf(stderr, "can't register RUSER service\n");
       exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

Figure 15-5 (Part 1 of 2). Using svc_register, a Low-Level RPC Routine

```
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;

    switch (rqstp->rq_proc) {
    case NULLPROC:
       if (!svc_sendreply(transp, xdr_void, 0)) {
          fprintf(stderr, "can't reply to RPC call\n");
          exit(1);
       }
       return;
    case RUSERSPROC_NUM:
       /*
        * code here to compute the number of users
        * and put in variable nusers
        */
       if (!svc_sendreply(transp, xdr_u_long, &nusers) {
          fprintf(stderr, "can't reply to RPC call\n");
          exit(1);
       }
       return;
    default:
       svcerr_noproc(transp);
       return;
    }
}
```

Figure 15-5 (Part 2 of 2). Using svc_register, a Low-Level RPC Routine

First, the server gets a transport handle, which is used for sending out RPC messages. The **registerrpc** routine uses the **svcudp_create** routine to get a UDP/IP handle. If you require a reliable protocol, you should call the **svctcp_create** routine. If the argument to the **svcudp_create** routine is RPC_ANYSOCK, the RPC library creates the socket. If you specify a different socket, it can be either bound or unbound. If you bind the socket to a port, the port numbers of **svcudp_create** and **clntudp_create** must match.

When you make a **clntudp_create** call with an unbound socket, the system gets the port number from the portmap on the machine you are calling. The **portmap** is a daemon that keeps track of the port numbers of all registered RPC services and the servers that can receive them. If the portmap is not running or does not have a port that matches the remote procedure call, the call fails.

Note: You can call the **portmap** directly if you prefer. Search the **<rpc/pmap_prot.h>** header file for the procedure numbers required to call the portmap.

After creating a server transport handle, call the **pmap_unset** routine to erase the last entry for the program number associated with this remote procedure from the portmap tables so the procedure is identified by its new number.

Next, the program number for the **nusers** routine is associated with the procedure **nuser**. The last argument to **svc_register** is usually the protocol that is being used. The default value is IPPROTO_UDP so you must change it to IPPROTO_TCP if you have created a new TCP/IP socket. The IPPROTO_UDP and IPPROTO_TCP are constants set in the <netinet/in.h> file.

The user routine **nuser** must call and dispatchthe appropriate XDR routines based on the procedure number. The user service routine **nuser** serializes the results and returns them to the caller using the **svc_sendreply** routine.

See the RPC Service Routines section of *AIX Operating System Technical Reference* for information on RPC routines.

The Client Side of Low-Level RPC

On the client side, you can control the socket used to transport RPC data and the delivery protocol by using the **clnt_call** routine. This contrasts with using the **callrpc** routine in which you have no control over the parameters that set the socket and delivery protocol.

The **clnt_call** routine takes a pointer to a client. You can supply values to the following parameters of **clnt_call**:

- Client handle
- Procedure number associated with the remote procedure
- XDR routine for serializing the parameters
- Pointer to the location of the parameters
- XDR routine for deserializing the return value
- Pointer to location the return value is placed
- Time, in seconds, to wait for a reply.

The pointer to the client contains the transport protocol value. The **callrpc** routine automatically calls the **clntudp_create** routine to get the pointer. To specify TCP/IP as the delivery protocol, use the **clnttcp_create** routine.

The parameters to the **clntudp_create** routine that can be changed are as follows:

- Address and length of the server
- Remote procedure's program number
- Remote procedure's version number
- Amount of time the client waits between tries
- Pointer to the socket.

The number of times the **clnt_call** routine tries to reach the server is determined by dividing the amount of time the client waits between tries by the **clntudp_create** timeout value.

Include the **clnt_destroy** routine to deallocate space associated with the client's handle. It does not close the socket that was passed as a parameter to the **clntudp_create** routine. If there are multiple client handles using the same socket, you can close one handle without destroying the socket used by other handles.

Figure 15-6 on page 15-27 shows the low-level RPC code to call the **nusers** service from the server in Figure 15-5 on page 15-23.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int addrlen, sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    unsigned long nusers;

    if (argc < 2) {
       fprintf(stderr, "usage: nusers hostname-n");
       exit(-1);
    }
    if ((hp = gethostbyname(argv[1])) == NULL) {
       fprintf(stderr, "can't get addr for %s-n",argv[1]);
       exit(-1);
```

Figure 15-6 (Part 1 of 2). Low-Level RPC Client Passing an RPC Program using UDP/IP

```
    }
    pertry_timeout.tv_sec = 3;
    pertry_timeout.tv_usec = 0;
    addrlen = sizeof(struct sockaddr_in);
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
       hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clntudp_create(&server_addr, RUSERSPROG,
      RUSERSVERS, pertry_timeout, &sock)) == NULL) {
       clnt_pcreateerror("clntudp_create");
       exit(-1);
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_ void,
       0, xdr_u_long, &nusers, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
       clnt_perror(client, "rpc");
       exit(-1);
    }
    clnt_destroy(client);
}
```

Figure 15-6 (Part 2 of 2). Low-Level RPC Client Passing an RPC Program using UDP/IP

If you want to make a stream connection, replace the **clntudp_create** routine with the **clnttcp_create** routine. Code the new entry as follows:

```
clnttcp_create (&server_addr, prognum, versnum,
              &socket inputsize, outputsize);
```

Notice that in the **clnttcp_create** routine, there are no *timeout* parameters to supply, but you do need to supply values for the send and receive buffers. Once the TCP/IP connection is established, all remote procedure calls using the client handle pointed to by this creation routine use this socket. The server that answers a remote procedure call using TCP/IP has its **svcudp_create** routine replaced by the **svctcp_create** routine.

See the RPC Service Routines section of *AIX Operating System Technical Reference* for information about RPC library routines.

Authenticating Remote Procedure Calls

RPC servers are configured to contain the information to authenticate all remote procedure calls made with RPC routines. When remote procedure calls are made, the caller sends authentication parameters with the routine as identification. ***Authentication*** is a means of verifying the user of an information system or resource. You can associate different forms, sometimes called flavors, of authentication with RPC clients just as you can set different transport protocols (UDP/IP and TCP/IP) for message delivery when you create RPC clients and servers. The default authentication parameter is the null value or can be set by the **authunix_none** routine.

Authenticating the Client

When a caller creates a new RPC client handle, the routine returns the authentication handle declaration as the following:

```
clnt->cl_auth = authnone_create ();
```

You can set the RPC client authentication by setting the statement after creating the RPC client handle to the appropriate routine. For example, you can set the client authentication to be the default AIX style by the following entry:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with the client handle specified by **clnt** to carry the following authentication credentials structure:

```
/*
 * Unix style credentials.
 */
struct authunix_parms {
        u_long aup_time;          /* credentials creation time */
        char  *aup_machname;      /* host name where client is */
        int    aup_uid;           /* client's UNIX effective uid */
        int    aup_gid;           /* client's current group id */
        u_int  aup_len;           /* element length of aup_gids */
        int   *aup_gids;          /* array of groups user is in */
};
```

Figure 15-7. Example of Authentication Credentials Structure

The fields in the credentials structure are set when the **authunix_create_default** routine invokes the appropriate system calls.

The user who created the RPC authentication is responsible for destroying it to conserve memory. Use the following routine to free space used by the authentication:

```
auth_destroy(clnt->cl_auth);
```

Authenticating the Server

The RPC service package passes authentication parameters as opaque data. This means that the structures are passed as arbitrary types without interpretations. To understand the authentication data that is passed, examine the typical structure of an RPC request that follows:

Note: The comments to the side indicate the definition of the parameter.

```
struct svc_req {
        u_long rq_prog;         /* service program number */
        u_long rq_vers;         /* service protocol version number */
        u_long rq_proc;         /* desired procedure number */
        struct opaque_auth
                rq_cred;         /* raw credentials from wire */
        caddr_t rq_clntcred;  /* credentials that specify read only */
};
```

The **rq_cred** parameters are mostly opaque to the programmer. The exception is the style of authentication used.

The following fields can be set by the programmer:

```
struct opaque_auth {
        enum_t  oa_flavor;       /* style of credentials */
        caddr_t oa_base;         /* address of more auth stuff */
        u_int   oa_length;       /* not to exceed MAX_AUTH_BYTES */
};
```

The service implementer can inspect the request's **rq_cred.oa_flavor** to determine which style of authentication the caller used. The request's **rq_clntcred** field is **NULL** or points to a structure that corresponds to a supported style of authentication credentials.

Consider the following regarding server authentication:

- It is not necessary to check the authentication parameters associated with the **nullproc** procedure number since it means the procedure is number zero.
- If the authentication parameter's type cannot be matched on your server, call the **svcerr_weakauth** routine.
- The service protocol itself should return the information regarding denied access.
- RPC does not use an individual service's access controls. Each service must implement its own access control policies and reflect these policies as a return value in its protocol.

The remote users service example shown in Figure 15-8 on page 15-33 computes the results from the remote procedure call that requests the total number of remote users minus the user identified by UID 16.

```
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    struct authunix_parms *unix_cred;
    int uid;
    unsigned long nusers;

    /*
     * we don't care about authentication for null proc
     */
    if (rqstp->rq_proc == NULLPROC) {
       if (!svc_sendreply(transp, xdr_void, 0)) {
          fprintf(stderr, "can't reply to RPC call\n");
          exit(1);
       }
       return;
    }
    /*
     * now get the uid
     */
    switch (rqstp->rq_cred.oa_flavor) {
    case AUTH_UNIX:
       unix_cred = (struct authunix_parms *)rqstp->rq_clntcred;
       uid = unix_cred->aup_uid;
       break;
```

Figure 15-8 (Part 1 of 2). Remote Users Service Example

```
    case AUTH_NULL:
    default:
        svcerr_weakauth(transp);
        return;
    }
    switch (rqstp->rq_proc) {
    case RUSERSPROC_NUM:
        /*
         * make sure caller is allowed to call this proc
         */
        if (uid == 16) {
            svcerr_systemerr(transp);
            return;
        }
        /*
         * code here to compute the number of users
         * and put in variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers) {
            fprintf(stderr, "can't reply to RPC call\n");
            exit(1);
        }
        return;
    default:
        svcerr_noproc(transp);
        return;
    }
}
```

Figure 15-8 (Part 2 of 2). Remote Users Service Example

Special Topics

This section discusses certain aspects related to RPC functions.

The select Procedure on the Server Side

A **select** procedure allows you to ***poll*** devices, that is, read to see if there is data present and continue regular processing if not. For example, a process can execute RPC requests while periodically interrupting regular processing to update a data structure. The process sets an alarm signal before calling the **svc_run** routine. If the process' other activity involves waiting for a file descriptor, the **svc_run** call does not work. Figure 15-9 shows the code for **svc_run**:

```
void
svc_run()
{
    int readfds;

    for (;;) {
        readfds = svc_fds;
        switch (select(32, &readfds, NULL, NULL, NULL)) {

        case -1:
            if (errno == EINTR)
                continue;
            perror("rstat: select");
            return;
        case 0:
            break;
        default:
            svc_getreq(readfds);
        }
    }
}
```

Figure 15-9. Code for the Library Routine svc_run

If you identify the file descriptors of the sockets for the programs for which you are waiting, you can bypass the **svc_run** routine and call the **svc_getreq** routine directly.

See the RPC Service Routines section in *AIX Operating System Technical Reference* for information on RPC library routines.

Broadcast RPC

With RPC, you can broadcast remote procedure calls across the network. When you broadcast data, it is transmitted to all servers listening at the port or ports the data is transmitted on. This means clients do not have to depend on a particular server to answer requests. Clients can also broadcast remote procedure calls in order to receive more than one answer to a service request.

Broadcast RPC is only supported by transport protocols that deliver information in packet form, such as UDP/IP.

Broadcast RPC filters out all unsuccessful responses. The user does not know if calls were discarded or not. For example, if a mismatch between the broadcast and remote service versions occurs, the responses are filtered out.

Broadcast messages are sent through the **portmap** daemon. The ***portmap*** daemon program, sometimes called the **portmapper** runs on the local servers to map RPC program program numbers to the ports that service each procedure. RPC can only access broadcast service requests that are registered with their **portmap** daemon. The **portmap** daemon, in conjunction with standard RPC protocols, is required to use broadcast RPC.

Figure 15-10 shows an example of broadcast RPC:

```
#include <rpc/pmap_clnt.h>

enum clnt_stat  clnt_stat;

clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults,
    resultsp, eachresult)
u_long          prog;        /* program number */
u_long          vers;        /* version number */
u_long          proc;        /* procedure number */
xdrproc_t       xargs;       /* xdr routine for args */
caddr_t         argsp;       /* pointer to args */
xdrproc_t       xresults;    /* xdr routine for results */
caddr_t         resultsp;    /* pointer to results */
bool_t (*eachresult)();      /* call with each result gotten */
```

The procedure **eachresult** is called for each valid result. It returns a boolean expression that indicates whether or not the client wants more responses, as the following example shows:

```
bool_t                  done;
done =
eachresult(resultsp, raddr)
caddr_t resultsp;
struct sockaddr_in *raddr;  /* addr of responding machine */
```

If done is TRUE, broadcasting stops and **clnt_broadcast** returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after waiting a few seconds. If no response comes back, the routine returns with RPC_TIMEDOUT. To interpret **clnt_stat** errors, feed the error code to the **clnt_perrno** routine.

Using Batching

When you use RPC, the client sends a call message and waits for servers to reply. RPC batch facilities allow clients to continue processing while waiting for a response from a server.

When you use batching, you can place RPC messages in a pipeline of calls to a server. The pipeline of calls is carried on a reliable transport such as TCP/IP. RPC batching is more efficient than single call messages because:

- The client can continue to generate new calls while the server executes previous ones.
- TCP/IP can send many call messages to the server in one **write** system call.

Note: No response is required from the server for each call in the pipeline.

Figure 15-10 shows an example of RPC batching with TCP/IP. It has two calls that create strings. One call returns void results. The other call does not return results.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL){
       fprintf(stderr, "can't create an RPC server-n");
       exit(1);
```

Figure 15-10 (Part 1 of 3). Using RPC Batching with TCP/IP Delivery

```
}
pmap_unset(WINDOWPROG, WINDOWVERS);
if (!svc_register(transp, WINDOWPROG, WINDOWVERS,
      windowdispatch, IPPROTO_TCP)) {
        fprintf(stderr, "can't register WINDOW service-n");
        exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point-n");
}
void
windowdispatch(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    char *s = NULL;

    switch (rqstp->rq_proc) {
    case NULLPROC:
       if (!svc_sendreply(transp, xdr_void, 0)) {
          fprintf(stderr, "can't reply to RPC call-n");
          exit(1);
       }
       return;
    case RENDERSTRING:
       if (!svc_getargs(transp, xdr_wrapstring, &s)) {
          fprintf(stderr, "can't decode arguments-n");
          svcerr_decode(transp);
          break;
}
```

Figure 15-10 (Part 2 of 3). Using RPC Batching with TCP/IP Delivery

```
        /*
         * call here to render the string s
         */
        if (!svc_sendreply(transp, xdr_void, NULL)) {
            fprintf(stderr, "can't reply to RPC call-n");
            exit(1);
        }
        break;
    case RENDERSTRING_BATCHED:
        if (!svc_getargs(transp, xdr_wrapstring, &s)) {
            fprintf(stderr, "can't decode arguments-n");
            /*
             * we are silent in the face of protocol errors
             */
            break;
        }
        /*
         * call here to render string s, but send no reply!
         */
        break;
    default:
        svcerr_noproc(transp);
        return;
    }
    /*
     * now free string allocated while decoding arguments
     */
    svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Figure 15-10 (Part 3 of 3). Using RPC Batching with TCP/IP Delivery

The server can have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

For a client to take advantage of batching, it must perform RPC calls on a TCP-based delivery. The calls must have the following attributes:

- The result of the XDR routine must be zero (**NULL**).
- The RPC call timeout must be zero.

Figure 15-11 shows an example of a client using batching to create several strings. The batch is emptied when the client gets a null string.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>

main(argc, argv)
    int argc;
    char **argv;
{
    struct hostent *hp;
    struct timeval pertry_timeout, total_timeout;
    struct sockaddr_in server_addr;
    int addrlen, sock = RPC_ANYSOCK;
    register CLIENT *client;
    enum clnt_stat clnt_stat;
    char buf[1000], *s = buf;

    /* initial as in Figure 15-6 on page 15-27
     */
    if ((client = clnttcp_create(&server_addr,
      WINDOWPROG, WINDOWVERS, &sock, 0, 0)) == NULL) {
        perror("clnttcp_create");
        exit(-1);
```

Figure 15-11 (Part 1 of 2). Example of a Client Using Batching to Create Strings

```
    }
    total_timeout.tv_sec = 0;
    total_timeout.tv_usec = 0;
    while (scanf("%s", s) != EOF) {
        clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
            xdr_wrapstring, &s, NULL, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
            clnt_perror(client, "batched rpc");
            exit(-1);
        }
    }
    /* now flush the pipeline
     */
    total_timeout.tv_sec = 20;
    clnt_stat = clnt_call(client, NULLPROC, xdr_void, NULL,
        xdr_void, NULL, total_timeout);
    if (clnt_stat != RPC_SUCCESS) {
        clnt_perror(client, "rpc");
        exit(-1);
    }
    clnt_destroy(client);
}
```

Figure 15-11 (Part 2 of 2). Example of a Client Using Batching to Create Strings

The server does not send a message to notify clients of failures.

Using the inetd Daemon to Start a Server

You can start an RPC server from the **inetd** daemon. Change the original code to call the **svcudp_create** routine as follows:

```
transp = svcudp_create (0);
```

This way the **inetd** process passes a socket as a file descriptor with the value 0.

In addition, call the **svc_register** routine as:

```
svc_register (transp, PROGNUM, VERSNUM, service, 0);
```

with the final flag as the value 0, since the program is already registered by the **inetd** daemon.

Remember that if you want to exit from the server process and return control to the **inetd** daemon, you must explicitly exit because the **svc_run** routine never returns.

The format of entries in the **/etc/inetd.conf** file for RPC service procedures is as follows:

svc_name sunrpc_udp udp wait|nowait root *program prognum versnum*

The *program* field is the C code implementing the server, and *prognum* and *versnum* are the program and version numbers of the RPC procedure. The value udp can be replaced by tcp for TCP/IP based RPC services as follows:

svc_name sunrpc_tcp tcp nowait

The version number can be a range if the same program handles multiple versions, as shown in the following example:

```
rstatd sunrpc_udp udp wait root /usr/etc/rpc.rstadt rstatd 100001 1-2
```

Example Applications

The following sections show examples of specific network applications using RPC routines. The explanation accompanying each example briefly explains the use of each application. For a more detailed explanation of the routines and their parameters, see *AIX Operating System Technical Reference.*

Example of Using Version Numbers

An RPC version number identifies the version of an RPC program. It is possible to use several versions of the same program through a server in order to call a remote procedure. The following example shows how an RPC server can service two versions of the same program. In this example, the first version number of the program called RUSERSPROG is RUSERSVERS_ORIG. There is a new version of RUSERPROG that returns an unsigned short rather than an unsigned long. This version is named RUSERSVERS_SHORT. A server that supports both versions does a double register, as shown in Figure 15-12.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG,
  nuser, IPPROTO_TCP)) {
    fprintf(stderr, "can't register RUSER service-n");
    exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT,
  nuser, IPPROTO_TCP)) {
    fprintf(stderr, "can't register RUSER service-n");
    exit(1);
}
```

Figure 15-12. A Server Supporting Two Versions of the Same Program

A C program can be written to handle both versions. Figure 15-13 on page 15-46 shows an example of a C program that handles both versions.

```
nuser(rqstp, tranp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    unsigned long nusers;
    unsigned short nusers2;

    switch (rqstp->rq_proc) {
    case NULLPROC:
       if (!svc_sendreply(transp, xdr_void, 0)) {
          fprintf(stderr, "can't reply to RPC call-n");
          exit(1);
       }
       return;
    case RUSERSPROC_NUM:
       /*
        * code here to compute the number of users
        * and put in variable nusers
        */
       nusers2 = nusers;
       if (rqstp->rq_vers != RUSERSVERS_ORIG)
          return;
       if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
          fprintf(stderr, "can't reply to RPC call-n");
          exit(1);
```

Figure 15-13 (Part 1 of 2). C Procedure Handling Two Versions of the Same Program

```
        } else if (!svc_sendreply(transp, xdr_u_short, &nusers2)) {
            fprintf(stderr, "can't reply to RPC call-n"); exit(1);
        } return;
    default:
        svcerr_noproc(transp); return;
    }
}
```

Figure 15-13 (Part 2 of 2). C Procedure Handling Two Versions of the Same Program

Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

In the following example, RPC uses the Transmission Control Protocol/Internet Protocol (TCP/IP) to carry messages between communicating programs. TCP/IP transports RPC messages in long streams of data. In Figure 15-14 on page 15-48, the initiator of the remote procedure call takes its standard input and sends it to the server. The server lists it to standard output. This example also shows an XDR procedure that behaves differently on serialization than on deserialization.

```
/*
   * The xdr routine:
   *              on decode, read from wire, write onto fp
   *              on encode, read from fp, write onto wire
   */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
     XDR *xdrs;
     FILE *fp;
{
     unsigned long size;
     char buf[BUFSIZ], *p;

     if (xdrs->x_op == XDR_FREE)/* nothing to free */
        return 1;
     while (1) {
        if (xdrs->x_op == XDR_ENCODE) {
           if ((size = fread(buf, sizeof(char), BUFSIZ,
             fp)) == 0 && ferror(fp)) {
               fprintf(stderr, "can't fread-n");
               exit(1);
                        }
                    }
                    p = buf;
                    if (!xdr_bytes(xdrs, &p, &size, BUFSIZ))
                        return 0;
                    if (size == 0)
                        return 1;
                    if (xdrs->x_op == XDR_DECODE) {
```

Figure 15-14 (Part 1 of 5). Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

```
            if (fwrite(buf, sizeof(char), size,
              fp) != size) {
                fprintf(stderr, "can't fwrite-n");
                exit(1);
            }
        }
    }
}
/*
  * The sender routines
  */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/time.h>

main(argc, argv)
    int argc;
    char **argv;
{
    int err;

    if (argc < 2) {
        fprintf(stderr, "usage: %s servername-n", argv[0]);
        exit(-1);
    }
    if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP,
      RCPVERS, xdr_rcp, stdin, xdr_void, 0) != 0)) {
```

Figure 15-14 (Part 2 of 5). Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

```
        clnt_perrno(err);
        fprintf(stderr, "can't make RPC call-n");
        exit(1);
    }
}
callrpctcp(host, prognum, procnum, versnum,
          inproc, in, outproc, out)
    char *host, *in, *out;
    xdrproc_t inproc, outproc;
{
    struct sockaddr_in server_addr;
    int socket = RPC_ANYSOCK;
    enum clnt_stat clnt_stat;
    struct hostent *hp;
    register CLIENT *client;
    struct timeval total_timeout;

    if ((hp = gethostbyname(host)) == NULL) {
        fprintf(stderr, "can't get addr for '%s'-n", host);
        exit(-1);
    }
    bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr,
        hp->h_length);
    server_addr.sin_family = AF_INET;
    server_addr.sin_port =  0;
    if ((client = clnttcp_create(&server_addr, prognum,
      versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
        perror("rpctcp_create");
        exit(-1);
```

Figure 15-14 (Part 3 of 5). Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

```
    }
    total_timeout.tv_sec = 20;
    total_timeout.tv_usec = 0;
    clnt_stat = clnt_call(client, procnum,
       inproc, in, outproc, out, total_timeout);
    clnt_destroy(client)
    return (int)clnt_stat;
}
/*
  * The receiving routines
  */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
    register SVCXPRT *transp;

    if ((transp = svctcp_create(RPC_ANYSOCK,
      BUFSIZ, BUFSIZ)) == NULL) {
        fprintf("svctcp_create: error-n");
        exit(1);

    }
    pmap_unset(RCPPROG, RCPVERS);
    if (!svc_register(transp,
      RCPPROG, RCPVERS, rcp_service, IPPROTO_TCP)) {
        fprintf(stderr, "svc_register: error-n");
        exit(1);
```

Figure 15-14 (Part 4 of 5). Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

```
    }
    svc_run);  /* never returns */
    fprintf(stderr, "svc_run should never return-n");
}
rcp_service(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
    case NULLPROC:
       if (svc_sendreply(transp, xdr_void, 0) == 0) {
          fprintf(stderr, "err: rcp_service");
          exit(1);
       }
       return;
    case RCPPROC_FP:
       if (!svc_getargs(transp, xdr_rcp, stdout)) {
          svcerr_decode(transp);
          return;
       }
       if (!svc_sendreply(transp, xdr_void, 0)) {
          fprintf(stderr, "can't reply-n");
          return;
       }
       exit(0);
    default:
       svcerr_noproc(transp);
       return;
    }
}
```

Figure 15-14 (Part 5 of 5). Example of Using Transmission Control Protocol/Internet Protocol (TCP/IP)

Example of Using Callback Procedures

A server can become a client and make a remote procedure call back to a client process. An RPC callback requires a program number on which to make the RPC call. Figure 15-15 shows how the **gettransient** routine generates a program number dynamically. The **gettransient** routine returns a valid program number in the transient range (see the table in "Assigning Program Numbers to Protocols" on page 15-5), and registers the number with the **portmap** daemon, sometimes called the **portmapper**. It communicates only with the **portmap** daemon running on the same machine as the **gettransient** routine. The call to **pmap_set** is a test and set operation that tests whether a program number has already been registered, and reserves the number if not.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
    int proto, vers, *sockp;
{
    static int prognum = 0x40000000;
    int s, len, socktype;
    struct sockaddr_in addr;
    switch(proto) {
        case IPPROTO_UDP:
            socktype = SOCK_DGRAM;
            break;
        case IPPROTO_TCP:
            socktype = SOCK_STREAM;
            break;
        default:
```

Figure 15-15 (Part 1 of 2). Example of Using a Callback Procedure

```
            fprintf(stderr, "unknown protocol type-n");
            return 0;
        }
        if (*sockp == RPC_ANYSOCK) {
            if ((s = socket(AF_INET, socktype, 0)) < 0) {
                perror("socket");
                return (0);
            }
            *sockp = s;
        }
        else
            s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);
        /*
         * may be already bound, so don't check for error
         */
        bind(s, &addr, len);
        if (getsockname(s, &addr, &len)< 0) {
            perror("getsockname");
            return (0);
        }
        while (!pmap_set(prognum++, vers, proto, addr.sin_port))
            continue;
        return (prognum-1);
}
```

Figure 15-15 (Part 2 of 2). Example of Using a Callback Procedure

The programs shown in Figure 15-16 on page 15-55 and Figure 15-17 on page 15-57 show how to make an RPC callback using the **gettransient** routine. The client makes an RPC call to the server, passes a transient program number, and waits to receive a callback from the server at that program number. The server registers the program EXAMPLEPROG, to receive the RPC call informing it of the callback program number. At some time (which in

this example is when it receives SIGALRM), it sends a callback RPC call using the program number it received earlier.

```
/*
  * client
  */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
    char **argv;
{
    int x, ans, s;
    SVCXPRT *xprt;

    gethostname(hostname, sizeof(hostname));
    s = RPC_ANYSOCK;
    x = gettransient(IPPROTO_UDP, 1, &s);
    fprintf(stderr, "client gets prognum %d-n", x);
    if ((xprt = svcudp_create(s)) == NULL) {
      fprintf(stderr, "rpc_server: svcudp_create-n");
        exit(1);
    }
    /* protocol is 0 - gettransient() does registering
     */
    (void)svc_register(xprt, x, 1, callback, 0);
    ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEVERS,
       EXAMPLEPROC_CALLBACK, xdr_int, &x, xdr_void, 0);
```

Figure 15-16 (Part 1 of 2). Using the gettransient Routine with a Client

```
    if (ans != RPC_SUCCESS) {
        fprintf(stderr, "call: ");
        clnt_perrno(ans);
        fprintf(stderr, "-n");
    }
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return-n");
}
callback(rqstp, transp)
    register struct svc_req *rqstp;
    register SVCXPRT *transp;
{
    switch (rqstp->rq_proc) {
        case 0:
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: rusersd-n");
                exit(1);
            }
            exit(0);
        case 1:
            if (!svc_getargs(transp, xdr_void, 0)) {
                svcerr_decode(transp);
                exit(1);
            }
            fprintf(stderr, "client got callback-n");
            if (!svc_sendreply(transp, xdr_void, 0)) {
                fprintf(stderr, "err: rusersd");
                exit(1);
            }
    }
}
```

Figure 15-16 (Part 2 of 2). Using the gettransient Routine with a Client

```
/*
  * server
  */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;                /* program number for callback routine */

main(argc, argv)
    char **argv;
{
    gethostname(hostname, sizeof(hostname));
    registerrpc(EXAMPLEPROG, EXAMPLEVERS,
      EXAMPLEPROC_CALLBACK, getnewprog, xdr_int, xdr_void);
    fprintf(stderr, "server going into svc_run-n");
    signal(SIGALRM, docallback);
    alarm(10);
    svc_run();
    fprintf(stderr, "Error: svc_run shouldn't return-n");
```

Figure 15-17 (Part 1 of 2). Using the gettransient Routine with a Server

```
}

char *
getnewprog(pnump)
    char *pnump;
{
    pnum = *(int *)pnump;
    return NULL;
}

docallback()
{
    int ans;

    ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0,
        xdr_void, 0);
    if (ans != 0) {
        fprintf(stderr, "server: ");
        clnt_perrno(ans);
        fprintf(stderr, "-n");
    }
}
```

Figure 15-17 (Part 2 of 2). Using the gettransient Routine with a Server

Chapter 16. Message Facility

CONTENTS

About This Chapter 16-2
Message Catalogs 16-3
Message Text Source Files 16-5
 mkcatdefs 16-9
 runcat 16-11
Creating Catalogs 16-12
Using Catalogs 16-12
 Opening and Closing Catalogs 16-13
 Reading Messages 16-14
 Displaying Messages from a Shell Script 16-22

About This Chapter

This chapter includes a discussion of the AIX Message Facility, which allows messages displayed by an application program to be separated from that application. The Message Facility uses catalogs to store messages. This chapter describes the creation and use of message catalogs.

The AIX Message Facility allows you to separate the messages displayed by an application from the application itself. The message text is placed in a file called a ***message catalog***, from which the application reads and displays the messages as they are needed. This separation allows you to alter the message text without changing the application and recompiling. By using the message facility, you can translate messages into other languages without altering the application.

The AIX Message Facility includes interfaces for generating message catalogs and for accessing them from application programs. The interfaces and file formats used by the AIX Message Facility are a superset of standard message catalog utilities defined by the X/Open Group published in the *X/OPEN Portability Guide*. Applications that conform to X/Open should be compatible with the AIX Message Facility. AIX extensions to X/Open are noted throughout this chapter.

To create a message catalog, enter the text of your messages in a message source file. Process the message source file using the message facility's **gencat** program (see "Message Text Source Files" on page 16-5 and "Creating Catalogs" on page 16-12).

Before a message can be displayed, the message catalog must be opened. Use one of the message facility functions **catopen** or **NLcatopen** to open the catalog containing the message. "Opening and Closing Catalogs" on page 16-13 describes these functions. Next, use one of the message facility functions described in "Reading Messages" on page 16-14 to retrieve the message from the specified catalog. Use a standard output facility, such as **printf** or **NLprintf**, to display the message text.

The message facility also includes functions that can be called from a shell script to display messages contained in a message catalog. "Displaying Messages from a Shell Script" on page 16-22 discusses the use of these functions.

Message Catalogs

A message catalog is a file that contains messages used by an application. Each message in a message catalog is identified by a message set number and a message ID number. You assign these numbers to the messages when you enter the message text. You can use message set numbers to group related messages within a message catalog.

A message catalog can be virtually any size. The maximum numbers of sets in a catalog, of messages in a catalog, and of bytes in a message are defined in the file **/usr/include/limits.h** by three variables: **NL_SETMAX**, **NL_MSGMAX**, and **NL_TEXTMAX**, respectively.[1] A set number must be one or greater and less than or equal to **NL_SETMAX**. Similarly, a message ID number must be one or greater and less than or equal to **NL_MSGMAX**. If you exceed the **NL_SETMAX** or **NL_MSGMAX** limits, the **gencat** utility issues an error message and does not create or update the message catalog. Messages greater than **NL_TEXTMAX**, the maximum number of bytes in a message, are truncated to fit this limit, and a warning is issued by the **gencat** utility. The file **limits.h.** also contains the variable **NL_MAXOPEN**, which specifies the maximum number of catalogs that can be open at one time.

The message facility subroutines **catopen** and **NLcatopen**, which open the message catalogs, refer to the catalogs by file name (see "Opening and Closing Catalogs" on page 16-13). If the file does not exist where the ***absolute path*** specifies, an error occurs, causing **catopen** to fail. No error message is issued, and the program does not abort. An absolute path is a file name containing a / (slash).

If the catalog file name does not include a path, the user's environment determines the directory paths to search. Use the environment variable **NLSPATH** to define the path. You can use two special variables in **NLSPATH**: *%N* and *%L*. The *%N* string is replaced by the name of the catalog referred to in the function opening the message catalog, while *%L* is replaced by the value of the **LANG** environment variable.

You can use the **LANG** environment variable to refer to message catalogs that are separated into directories based on natural languages. For example, if **catopen** specifies a catalog named `mycmd`, and the environment variables are set as shown below:

```
$
NLSPATH=../%N:./%N:/system /nls/%L/%N:/system/nls/%N
$ LANG=kanji
```

the application will search for the catalog in the order shown here:

```
../mycmd
./mycmd
/system/nls/kanji/mycmd
/system/nls/mycmd
```

1 X/Open recommends the following minimums for these limits:

- A maximum set number no less than 255.
- A maximum message ID number no less than 32,767.
- A maximum number of bytes in a message no less than 255.

If no *%N* is present in a directory specification within **NLSPATH**, the application assumes the path is a directory and searches that directory for the catalog before searching the next path. The default value for **NLSPATH** is **/usr/lpp/msg/%L/%N**.

Message Text Source Files

A message text source file is a text file you create for messages. You can use any text editor to enter the messages. Assign message set numbers and message ID numbers to each message by using the **$set**, **$delete**, and **$quote** commands described in this section.

Use the **$set** command in a source file to give a group of messages a set number. The format of the **$set** command is:

```
$set n [comment]
```

The message set number is specified by *n*. All messages following the **$set** command are assigned that set number, until the next occurrence of a **$set** command. You must specify at least one set number if the source file contains message text. The set numbers must be assigned in ascending order, but need not be contiguous. Empty sets are created for skipped numbers. There is no performance advantage to the use of more than one set number in a catalog.

You can include an optional comment in the **$set** command. The following example includes a comment:

```
$set 10  Communication Error Messages
```

Use the **$delset** command to remove from a catalog all the messages belonging to the specified set. The format of the **$delset** command is:

```
$delset n [comment]
```

The message set is specified by *n*. The **$delset** command must be placed in the proper set-number order with respect to any **$set** commands in the same source file. You can include an optional comment in the **$delset** command.

You can include a comment line anywhere in the source file, except within message text. Indicate comments as shown below:

```
$ [comment]
```

You must leave at least one space after the $.

Enter the message text and symbolic message identifier as follows:

ID message-text

ID can be either a number or a symbolic identifier[2] and can contain only letters, digits, or the _ (underscore character). The maximum length of an identifier is 65 characters. The **mkcatdefs** program (see "mkcatdefs" on page 16-9) assigns a message number to the identifier. The assigned number is one higher than the preceding message number, or 1 if it is the first message after the `$set` command.

Note that **mkcatdefs** inserts a `$delset` before the `$set`, which means you cannot add, delete, or replace single messages in the catalog if you are using symbolic message identifiers. You must enter all messages in the set.

You can mix numbers and symbolic identifiers.

You must leave at least one space after the message identifier or number.[3] All text following the first nonblank character is included in the message text, to the end of the line. If the source contains a `$quote` command preceding the message, all text between the two quotation marks is included. Use the \ (escape character) to continue the message text on the following line. The \ must be the last character on the line, as in the following example:

```
FIVE     Text associated with \
message FIVE.
```

These two lines define the single-line message:

```
FIVE     Text associated with message FIVE.
```

The \ can be used to include special characters in the message text. These special characters are defined as follows:

\n	Performs a new-line function when the message is displayed.
\t	Inserts a horizontal tab character when the message is displayed.
\v	Inserts a vertical tab when the message is displayed.
\b	Performs a backspace function when the message is displayed.
\r	Inserts a carriage return character when the message is displayed.
\f	Inserts a form feed character when the message is displayed.
\\	Displays the backslash (\) character in the message.

2 A symbolic identifier is an extension to the X/Open specifications.

3 AIX allows any amount of white space after the message ID number; however X/Open specifies that you leave only one space between the message number and the message text.

ddd Displays the single-byte character associated with the octal value represented by the valid octal digits *ddd*. One, two, or three octal digits can be specified, but you must include leading zeros if the characters following the octal digits are also valid octal digits. For example, the octal value for **$** is **44**. To display $5.00 use \0445.00, not \445.00, or the 5 will be parsed as part of the octal value.

\x*dddd*[4] Displays the 1-byte or 2-byte character associated with the hexadecimal value represented by the four valid hexadecimal digits *dddd*. You can specify one, two, three, or four digits, but you must include leading zeros to avoid parsing errors (see *ddd*).

You can also include **printf** conversion specifications in messages that are displayed by applications using **printf** or **NLprintf** (see **printf** in *AIX Operating System Technical Reference*). If you display a message from a shell script with **dspmsg**, the message can contain the *%s* or *%n$s* conversion specifications (see "Displaying Messages from a Shell Script" on page 16-22).

You can use the **$quote** command in a message source file to define a character for delimiting message text. The format for this command is:

```
$quote [char] [comment]
```

Use the specified character before and after the message text as shown in the following example source file:

4 This escape sequence is an AIX extension to X/Open.

```
$quote "    Use a double quote to delimit message text

$set 10         Message Facility - symbolic identifiers

1   "Use the $quote command to define a character \
\n for delimiting message text."

2   "You can include the \"quote\"
character in a message \n \
by placing a \\ in front of it"

3     You can include the "quote" character in a
message \n \
by having another character as the first nonblank \
\n character after the message ID number

$quote "      Use double quotation marks to delimit message text

$delset 1
$set 1

1     "Symbolic identifiers can only contain alphanumeric \
characters or the _ (underscore character)\n"

2     "Symbolic identifiers cannot be more than 65 \
characters long\n"

5     "You can mix symbolic identifiers and numbers\n"

$quote

6     remember to include the "msg_h" file in your program
```

mkcatdefs

The **mkcatdefs**[5] message facility program changes symbolic identifiers in the message source file to a form that **gencat** can use. The **mkcatdefs** program produces the *symbname*_**msg.h** file. This file contains definition statements equating your symbolic identifiers with set numbers and message ID numbers assigned by **mkcatdefs**. After running **mkcatdefs** you can use symbolic names in an application to refer to messages. The *sourcefile* parameters specify the message source files containing the symbolic identifiers. The format for **mkcatdefs** is:

```
$ mkcatdefs catname sourcefile [sourcefile...] [-h]
```

After preprocessing a message source file, use **gencat** to create a message catalog named *catname*.

The **mkcatdefs** subroutine accepts multiple source files, but only one _**msg.h** file is produced. A **mkcatdefs** flag, _**h**, suppresses generation of the _**msg.h** file.

When using symbolic identifiers, you must include the **&catnmsgh.** definition file in your application program to refer to set numbers and message numbers assigned by **mkcatdefs**. The message source data, with numbers instead of symbolic identifiers, is sent to standard output. This output is suitable as input to the **gencat** program. You can use the > (redirection symbol) to write the message source to a file, then use that file as input to **gencat**. You can also use the **runcat** shell script described in "runcat" on page 16-11.

When you process this file with **mkcatdefs**, the modified source is written to standard output. Standard output can either be redirected to a file using the > (redirection symbol) or piped to **gencat** (see "Creating Catalogs" on page 16-12).

The following source is created:

5 **mkcatdefs** is an AIX extension to X/Open.

```
$quote "      Use double quotation marks to delimit message text

$delset 1
$set 1

1     "Symbolic identifiers can only contain alphanumeric \
characters or the _ (underscore character)\n"

2     "Symbolic identifiers cannot be more than 65 \
characters long\n"

5     "You can mix symbolic identifiers and numbers\n"

$quote

6     remember to include the "msg_h" file in your program
```

Note that the assigned message numbers are noncontiguous because the source contained a specific number. The **mkcatdefs** program always assigns the previous number plus 1 to a symbolic identifier.

The **mkcatdefs** program also produces a definition file for inclusion in your program. The name of the file is the **Symbname** entered as the first parameter to the **mkcatdefs** command. (If you specify the **-h** flag instead of the *symbname*, no definition file is produced.)

If the symbolic source defined above were in a file called `symb.src`, you could use the **mkcatdefs** command as follows:

```
$ mkcatdefs symb symb.src >symb.msg
```

The generated `symb_msg.h` file looks as follows:

```
#include <limits.h>
#include <nl_types.h>
#define MF_SYMB "symb.cat"


/* The following was generated from symb.src. */


/* definitions for set MSFAC */

#define SYM_FORM 1
#define SYM_LEN 2
#define MSG_H 6
```

You must include this definition file in your program to link the symbolic names to the set and message numbers assigned by **mkcatdefs**.

The **mkcatdefs** subroutine creates the symbol name `MF_SYMB` by making these changes: adding **MF_** to the name of the catalog specified by *catname*, using uppercase characters, and appending *.cat*. You can use this symbol in calls to **catopen** or **NLcatopen** instead of the catalog file name `symb`. Remember to use uppercase characters.

Since **mkcatdefs** includes **limits.h** and **nltypes.h**, you need not include them in your application program. (**nltypes.h** defines special data types required by the message facility routines.)

runcat

The AIX **runcat** message facility shell script invokes **mkcatdefs** and pipes the message source data (the output from **mkcatdefs**) to the **gencat** program.[6] This method is simpler than using the > (redirection symbol) to capture the output from **mkcatdefs** and running **gencat**. The format for **runcat** is:

```
$ runcat catname sourcefile [catfile]
```

The file *sourcefile* contains the message text with the symbolic identifiers. The **mkcatdefs** subroutine uses *catname* to generate the name of the symbolic definition file by adding *_msg.h* to the end of *catname*, and to generate the symbolic name for the catalog file by adding *MF_* to the beginning of *catname*. The definition file must be included in your

6 The shell script **runcat** is an AIX extension to X/Open.

application program. The symbolic name for the catalog file can be used in **catopen** or **NLcatopen** instead of the actual file name.

The catalog file created by **gencat** is called *catfile*. The **gencat** subroutine names the catalog file by adding *.cat* to the end of *catname*, unless you specify this parameter. This catalog file name must be used in **catopen** or **NLcatopen**.

Creating Catalogs

Use **gencat** to create a message catalog from a message text source file. "Message Text Source Files" on page 16-5 describes the format for a message source file. Because **gencat** conforms to X/Open specifications, it does not accept symbolic identifiers. You must run **mkcatdefs** as described in "Message Text Source Files" on page 16-5 if you want to use symbolic identifiers.

The format for **gencat** is:

```
$ gencat catfile [sourcefiles]
```

If a message catalog with the name *catfile* exists, **gencat** will modify the catalog according to the statements in the message source files. If a message catalog does not exist, **gencat** creates a catalog file with the name *catfile*.

You can specify any number of message text source files. Multiple files are processed in the sequence you specify. Each successive source file modifies the catalog. If you do not specify a source file, **gencat** accepts message source data from standard input.

Using Catalogs

Message catalogs can be accessed from application programs or shell scripts. This section describes the functions you use to access the messages in an existing catalog. To use the message facility functions described here, you must know the message set numbers and message ID numbers for the particular message text that you want to display. You must also know the catalog file that contains the messages. You can use symbolic identifiers for catalogs, sets, and messages, instead of file names and numbers if you use the symbolic message facilities described in "Message Text Source Files" on page 16-5.

The steps for using message catalogs from an application program are:

1. Open the catalogs that contain the required messages. "Opening and Closing Catalogs" on page 16-13 describes the message facility functions **catopen** and **NLcatopen**, which open catalogs.
2. Read the message text from the open catalog. "Reading Messages" on page 16-14 describes the message facility functions that read from a catalog.

3. Display the messages. AIX provides library functions, such as **printf** and **NLprintf**, that can be used to display messages. Use of these functions is described in the following sections where appropriate.

4. Close the catalog file. The function **catclose**, described in "Opening and Closing Catalogs," closes a particular catalog.

"NLgetamsg" on page 16-19 describes the function **NLgetamsg**, which opens a message catalog, retrieves a message, and closes the catalog each time it is called. **NLgetamsg** is an AIX extension to X/Open.

The AIX message facility includes functions which you can use from shell scripts to display messages. These functions are described in "Displaying Messages from a Shell Script" on page 16-22.

Opening and Closing Catalogs

You can use either **catopen** or **NLcatopen** to open a message catalog from within your application programs. The **catopen** subroutine is defined by X/Open. The **NLcatopen** subroutine is an AIX extension to X/Open.

catopen

The **catopen** subroutine opens a specified message catalog and returns a *catalog descriptor* which you use to retrieve messages from the catalog. The C language description for **catopen** is:

```
nl_catd catopen(catname,parm)
char *catname;
int parm;
```

nl_catd is a special data type for catalog descriptors. This data type is defined in the **nltypes.h** file; therefore, you must include this file in your application program. If you use symbolic identifiers and run **mkcatdefs**, **nltypes.h** is already included in the symbolic definition file (see "mkcatdefs" on page 16-9).

The *catname* file is the message catalog file that you open. The parameter *parm* is included for compatibility with X/Open, but is not used by AIX, and must have a value of zero.

If the catalog file name begins with a /, it is a directory search path. If the catalog file name is not a directory search path, the user's environment determines the directory paths to search (see "Message Catalogs" on page 16-3). If **catopen** cannot find the file, it returns -1 to indicate an error.

If the number of catalogs already open is equal to the **NL_MAXOPEN** limit defined in the file **limits.h**, **catopen** also returns -1. You must include **limits.h** in your application

program. If you use symbolic identifiers and run **mkcatdefs**, **limits.h** is already included in the symbolic definition file (see "mkcatdefs" on page 16-9).

NLcatopen

The **NLcatopen** subroutine does not open catalogs; instead, it prepares a catalog so that **NLcatgets** can open it the first time you want to retrieve a message (see "NLcatgets" on page 16-18). This method saves the overhead of opening a file that may not be read, or delays the opening of a file until a message is needed.

The **NLcatopen** subroutine also returns a catalog descriptor. The C language description for **NLcatopen** is:

```
nl_catd NLcatopen(catname)
char *catname;
```

The catalog file is called *catname*. If **NLcatopen** cannot find the file, or if the number of open catalogs is **NL_MAXOPEN**, **NLcatopen** returns -1.

catclose

If your program must access several message catalogs, you can reach the allowable number of open catalogs, and you must close some before opening more. It is also good programming practice to close any file you open. The **catclose** subroutine closes a specific catalog. The C language description for **catclose** is:

```
int catclose(catd)
nl_catd catd;
```

Either **catopen** or **NLcatopen** returns a catalog descriptor, *catd*. The **catclose** subroutine returns a 0 if it closes the catalog successfully and a -1 if it fails to close the catalog. If *catd* is not a valid catalog descriptor, **catclose** fails.

Reading Messages

The **catgets** and **catgetmsg** subroutines are defined by X/Open to retrieve messages from an open catalog. The **NLcatgets** and **NLgetamsg** subroutines are included in AIX, but are not specified by X/Open.

catgets

After a successful call to **catopen**, you can use the **catgets** subroutine to retrieve a message. The C language description for **catgets** is:

```
char *catgets(catd, set_num, msg_num, s)
nl_catd catd;
int set_num, msg_num;
char *s;
```

The **catopen** subroutine returns the *catd* catalog descriptor. The message set number, *set_num*, and the message ID number, *msg_num*, specify a particular message in the catalog.

If **catgets** finds the specified message, it loads that message into a character string buffer and terminates the message string with a null character. Then **catgets** returns a pointer to the buffer. You can use the pointer to reference the buffer and display the message. You can use **printf** or **NLprintf** with either the *%s* or *%n$s* conversion specification to display the message (see **printf** in *AIX Operating System Technical Reference*). The message in the buffer is overwritten by the next call to **catgets**.

If *catd* is not a valid catalog descriptor, **catgets** returns a pointer to the user-supplied error message string *s*. If **catgets** cannot find the specified message in the catalog, it returns a pointer to an empty string, that is, a string with length zero, terminated by a null character. Note: this is not the same as a null pointer.

catgetmsg

After a call to **catopen**, use **catgetmsg** to copy a message from a catalog into a user-defined character buffer. The C language description for **catgetmsg** is:

```
char *catgetmsg(catd, set_num, msg_num, buf, buflen)
nl_catd catd;
int set_num, msg_num, buflen;
char *buf;
```

The *catd* catalog descriptor returned from **catopen** specifies a catalog that **catgetmsg** accesses. The *set_num* and *msg_num* variables specify the message retrieved.

The **catgetmsg** subroutine returns the pointer *buf*, which points to a user-defined buffer if the message is found.

The length of the buffer is *buflen*. The subroutine **catgetmsg** copies up to *buflen-1* bytes of the message into the buffer specified by *buf*. The **catgetmsg** subroutine does not split a 2-byte character.

If **catgetmsg** fails for any reason, *buf* points to an empty string.

Creating a Message Catalog

The following example demonstrates the process of creating a message catalog from a source file and displaying the messages from a C language program. This example contains only the message facility functions defined by X/Open.

This is a message source file called `hello.txt`:

```
$ file: hello.txt

$set 1    prompts
1 Please, enter your name.
2 Hello, %s \n

$ end of file: hello.txt
```

Create a message catalog named `hello.cat` with **gencat** as follows:

```
$ gencat hello.cat hello.txt
```

The following C program, `hello`, illustrates opening the catalog `hello.cat` with **catopen**, retrieving messages from the catalog with **catgets** and **catgetmsg**, displaying the messages with **printf**, and closing the catalog with **catclose**.

```
/* program: hello */

#include <nltypes.h>
#include <limits.h>
#include <stdio.h>

nl_catd catd;

main()
{
   char name[25];
   char mybuf[NL_TEXTMAX+1];
   int buflen;
   /* open the catalog */
   if (catd=catopen("hello.cat",0)) == -1)
     printf("Can't open message catalog \n");

        /*The following tests whether a message exists in catalog.*/
        /*If yes, msg is displayed, otherwise default text.*/
   if (strlen(catgets(catd,1,1,"ENTER NAME: ")) == 0)
     printf("ENTER NAME: ");
   else printf(catgets(catd,1,1,"ENTER NAME: "));

   scanf("%s",name);                    /* read name */

   buflen = NL_TEXTMAX+1;

        /*The following tests whether a message was retrieved and */
        /*the catalog is open; if so greeting is displayed,  */
        /*otherwise, just name.                               */
   if((catd!=-1)&&(strlen(catgetmsg(catd,1,2,mybuf,buflen))!=0))
     printf(mybuf,name);               /* display greeting */
   else   printf("%s\n",name);       /* otherwise display */

   catclose(catd);                      /* close the catalog */
   exit(0);
}
```

In this example, **catopen** refers to `hello.cat` only by file name; therefore, you must make sure that the environment variable **NLSPATH** is set correctly, as discussed in "Message Catalogs" on page 16-3.

If you set the directory search path correctly, **catgets** returns a pointer to the specified message in the catalog `hello.cat`. If **NLSPATH** is not properly set, **catopen** returns a -1 for `catd`. Since this is an invalid catalog descriptor, **catgets** returns a pointer to the default string `ENTER NAME:`.

The second message from the catalog `hello.cat` has a *%s* conversion specification for the value of `name` to go into so that the message can be used as the format parameter in the **printf** subroutine.

If **catopen** cannot find the catalog, it returns a -1 for `catd`. This causes **catgetmsg** to return a pointer to an empty string (without a *%s*). If so, the program has a separate **printf** call to provide a *%s* for displaying the name.

NLcatgets

If you use **NLcatopen** to prepare a message catalog for access, you must use **NLcatgets** to access the catalog the first time. The C language description for **NLcatgets** is:

```
char *NLcatgets(catd, set_num, msg_num, s)
nl_catd catd;
int set_num, msg_num;
char *s;
```

The **NLcatopen** subroutine returns the *catd* catalog descriptor. The message set number, *set_num*, and the message ID number, *msg_num*, specify a particular message in the catalog.

If **NLcatgets** finds the specified message, it loads that message into a character string buffer and terminates the message string with a null character. **NLcatgets** returns a pointer to the buffer, which you can use to reference the buffer and display the message. The message in the buffer is overwritten by the next call to **NLcatgets**.

If **NLcatgets** fails for any reason, it returns a pointer to the user-supplied error message string *s*. The **NLcatgets** subroutine always returns a pointer to the catalog message or to the default message string *s*. Unlike the **catgets** subroutine, **NLcatgets** never points to a null string.

NLgetamsg

The **NLgetamsg** subroutine opens a specified catalog, retrieves a specified message, and closes the catalog, all in one call. This subroutine is particularly useful for infrequent message display.

The C language description for **NLgetamsg** is:

```
char *NLgetamsg(catname, set_num, msg_num, s)
char catname;
int set_num, msg_num;
char *s;
```

The **NLgetamsg** subroutine opens the message catalog called *catname*. The message set number *set_num* and the message ID number *msg_num* specify a particular message in the catalog. If **NLgetamsg** finds that message, the subroutine loads it into a character string buffer and terminates the message string with a null character. Then **NLcatgets** returns a pointer to the buffer.

If **NLgetamsg** fails for any reason, it returns a pointer to the user-supplied error message string *s*.

Creating a Message Catalog

This example shows how to create a message catalog from a source file with symbolic references, and how to use the AIX extensions to X/Open to retrieve the messages.

This is a message source file called `hello.sym` that contains symbolic references to the message set and the messages.

```
$ file: hello.sym
$quote "

$set PROMPTS
PLEASE  "Please, enter your name."
HELLO   "Hello, %s \n"

$ end of file: hello.sym
```

The following message source file contains error messages that can be referenced by their symbolic IDs:

```
$ file: msgerrs
$quote "

$set CAT_ERRORS
MAXOPEN  "Cannot open message catalog %s \n \
Maximum number of catalogs already open "
NOT_EX "File %s not executable \n "

$set MSG_ERRORS
NOT_FOUND "Message %1$d, Set %2$d not found \n "

$ end of file: msgerrs
```

Process the message source files `hello.sym` and `msgerrs` with **runcat** as follows:

```
$ runcat  hello  hello.sym

$ runcat  err  msgerrs  /usr/lpp/msg/english/msgerrs.cat
```

The **runcat** subroutine invokes **mkcatdefs** and **gencat**. The first call to **runcat** takes the source file `hello.sym` and uses the second parameter, `hello`, to produce a message catalog with the name `hello.cat` (see "runcat" on page 16-11), and a definition file with the name `hello_msg.h` (see "mkcatdefs" on page 16-9).

The definition file `hello_msg.h` contains a symbolic name for the message catalog (one that can be used in **NLcatopen**), as well as the symbolic IDs for the messages and sets. The symbolic name for `hello.cat` is `MF_HELLO`. This name is produced automatically by **mkcatdefs** (see "mkcatdefs" on page 16-9).

The second call to **runcat** takes the source file `msgerrs` and uses the second parameter, `err`, to produce a definition file with the name `err_msg.h`.

Since the third parameter, `/usr/lpp/msg/english/msgerrs.cat`, is present, **runcat** uses this parameter for the catalog file name. This parameter is an absolute file name which specifies exactly where **runcat** should put the file. The symbolic name for `msgerrs.cat` is `MF_MSGERRS`.

The C program `hello` is rewritten here to show symbolic references:

```
/* program: hello */

#include <nltypes.h>
#include <limits.h>
#include <stdio.h>
#include <err_msg.h>        /* symbolic definitions */
#include <hello_msg.h>

nl_catd catd;

main()
{
    char name[25];
    char *fmt;

    /* open the catalog */
    if (catd=NLcatopen(MF_HELLO.SYM,0) == -1)
        printf(NLgetamsg("/usr/lpp/msg/english/msgerrs.cat",
            CAT_ERRORS, MAXOPEN,
            "Can't open message catalog %s \n"),
            MF_HELLO);

    printf(NLcatgets(catd,PROMPTS,PLEASE,"ENTER NAME:"));

    scanf("%s",name);                    /* read name */

    /* display greeting, or if error, just the name */
    printf(NLcatgets(catd,PROMPTS,HELLO,"%s\n"),name);

    catclose(catd);                  /* close the catalog */
}
```

If the call to **NLcatopen** fails, the number of open message catalogs is already equal to the value of **NL_MAXOPEN**.

Displaying Messages from a Shell Script

AIX includes two extensions to X/Open for use in shell scripts: **dspcat** and **dspmsg**. You can also use these commands from the AIX command line. They use **NLSPATH**, as described in Chapter 16, "Message Facility" on page 16-1, to find a specified message catalog.

dspcat

Use **dspcat** to display a particular message, all of the messages in a set, or all of the messages in a catalog. The syntax for **dspcat** is:

```
dspcat catname [set_num] [msg_num] [-g]
```

set_num specifies a set in the catalog, and *msg_num* specifies a particular message in the set. If you include all three parameters, **dspcat** displays a particular message. If you do not include *msg_num*, all of the messages in the set are displayed. If you only specify *catname*, all of the messages in the entire catalog are displayed. You must include *set_num* if you include *msg_num*.

dspmsg

Use **dspmsg** to display a particular message from a catalog. You can pass up to ten string arguments for substitution into the message if it contains the **printf** conversion specification *%s*, or the **NLprintf** conversion specification *%n$s*. The syntax for **dspmsg** is:

```
dspmsg catname [-s set_num] msg_num ['default-message' [args]]
```

catname specifies a message catalog, *set_num* specifies a set in the catalog, and *msg_num* specifies a particular message in the set. The catalog (*&catname.*) and the message (*msg_num*) are required. The default set number is 1, but you can specify another set by using the *-s* flag followed by the set number.

If **dspcat** cannot find the message, the *default-message* is displayed. You must enclose the default message in single quotation marks if the default message contains %n$ format strings. If **dspcat** cannot find the message, and you do not specify a default message, a system-generated error message is displayed.

Follow the default message with up to ten arguments to substitute into the catalog message (or the default message). Missing arguments for conversion specifications are replaced by null strings.

Using dspcat

The following example shows how to use **dspcat** from the AIX command line to display messages from an existing catalog, and how to use **dspmsg** in a shell script to display specified messages from the same catalog. The message catalog generated from the message source file `msgerrs` in "Creating a Message Catalog" on page 16-19 is used in this example.

Use **dspcat** as follows to display all of the messages in the catalog.

```
/usr/lpp/msg/english/msgerrs.cat:

dspcat msgerrs
```

If you do not use a full path name, as in this example, be careful to set the **NLSPATH** environment variable so that **dspcat** searches the proper directory for the catalog (see "Message Catalogs" on page 16-3). The **LANG** environment variable can also affect the directory search path.

In this example, **dspcat** displays the entire contents of the catalog because no set or message IDs are specified. Here is the output:

```
1:1 Cannot open message catalog %s
Maximum number of catalogs already open
1:2 File %s not executable
2:1 Message %d, Set %d not found
```

This method allows you to see the symbolic ID numbers assigned to the message source `msgerrs` by **mkcatdefs**. You cannot readily use symbolic references with **dspmsg**, but using **dspcat** as shown can give you the necessary numbers.

The following is a simple shell script called `runtest` to show how to use **dspmsg**:

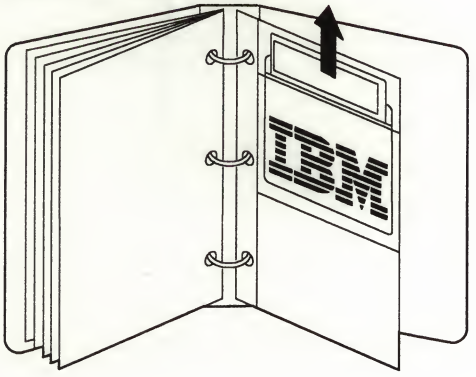
```
if [ - x ./test ]
    ./test;
else
    dspmsg -s 1 msgerrs.cat 2 '%s NOT EXECUTABLE \n' "test";
    exit;
```

Appendix A. Installing Programming Examples

This appendix explains how to install the Programming Examples. Before installing Programming Examples on the system, install the *IBM RT Virtual Resource Manager licensed program* and the *IBM RT AIX Operating System licensed program.*

Locating The Programming Examples Diskette

Remove the Programming Examples diskette from the plastic envelope in the back of the binder, as shown below:



A5AC6041

Installing Programming Examples from the Command Line

To install Programming Examples from the command line, use the steps that follow. If an error occurs during the procedure, see *IBM RT Messages Reference* for details.

How to Install Programming Examples

1. Log in as `su`, or as a member of the system group.
2. After the # prompt, type `installp`. Then press **Enter**.
3. Follow the prompts to insert the Programming Examples diskette and install the sample programs.
4. When installation is complete, remove the diskette and return it to the plastic envelope in the binder.
5. Log off the system.

More Detailed Information

1. You can log in to the system as `su`, or as a member of a system group. When you log in as `su`, you have the the effective user ID and privileges of the user `root`. See *Managing the AIX Operating System* for information about logging in as **su**.
2. Type `installp` command at the # prompt, then press **Enter**. The following prompt appears:

```
Insert the licensed program product volume 1 diskette
into diskette drive 0 and then press Enter.
```

3. Insert the Programming Examples diskette into the diskette drive, and press **Enter**. The following prompt appears:

```
Beginning installation of the licensed program product
      "Programming Examples"

Type "y" to continue or "n" to cancel, then press Enter.

--> y
```

4. Type `y` to continue the installation, and press **Enter**. The following message appears:

```
Installation of the licensed program product
      "Programming Examples"
is in progress.
```

 When installation is complete, the following message appears:

```
Installation of the licensed program product
      "Programming Examples"
was successfully completed.
```

5. Remove the Programming Examples diskette from the diskette drive. Return it to the plastic envelope in the binder or other safe place.
6. Log off the system.

Use the Programming Examples as directed in this book.

Appendix B. Extended curses Structures

WINDOW Structure

The ***extended curses*** library routines use a structure, ***WINDOW***, to hold information about each window that it is working with. Figure B-1 on page B-2 shows the contents of that structure.

```
struct _win_st
{
   short    _cury, _curx;
   short    _maxy, _maxx;
   short    _begy, _begx;
   short    _winy, _winx;
   short    _flags;
   short    *_firstch;
   short    *_lastch;
   bool     _clear;
   bool     _leave;
   bool     _scroll;
   ATTR     _csbp;
   NLSCHAR  **_y;
   ATTR     **_a;
   struct   _win_st *_view;
} ;

#define     WINDOW struct  _win_st
#define     _SUBWIN        001
#define     _ENDLINE       002
#define     _FULLWIN       004
#define     _SCROLLWIN     010
#define     _ISVIEW        040
#define     _HASVIEW       100
#define     _STANDOUT      200
#define     _NOCHANGE       -1
```

Figure B-1. Structure Definition for WINDOW

The variables in this structure perform the following functions:

_cury and _curx
The current (y, x) coordinates for the window. New characters added to the screen are added at this point.

_maxy and _maxx
One more than the maximum values allowed for _cury and _curx.

_begy and _begx
The starting (y, x) coordinates on the terminal for the window (the home position for the window). The variables _cury, _curx, _maxy, and _maxx are measured relative to _begy and _begx, not the home position for the terminal.

_winy, _winx
The starting (y, x) coordinates of a viewport within the original window.

_flags
A flag byte that can have one or more of the following values ORed into it:

_SUBWIN
Indicates that the window is a subwindow. The **delwin()** call checks this flag. If this flag is set, the space for the lines is not freed when the window is deleted.

_END-LINE
Indicates that the end of the line for this window is also the end of a screen.

_FULLWIN
Indicates that this window is a full screen window.

_SCROLLWIN
Indicates that the last character of this screen is at the lower right corner of the terminal. If a character is written to the lower right corner of the terminal, the terminal (hardware) scrolls automatically.

_ISVIEW
Indicates that the window is a viewport window.

_HASVIEW
Indicates that the window has a viewport window in it.

_STANDOUT
Indicates that all characters added to the screen are in standout mode.

***_firstch**
Pointer to the first position (row by row) in the optimization array that was changed. If this pointer contains the value _NOCHANGE, then a change was not made to a line since the last time that **refresh()** changed **curscr**.

***_lastch**

Pointer to the last position (row by row) in the optimization array that has been changed.

_clear

Tells if a clear-screen sequence is to be generated on the next **refresh()** call. This is only meaningful for screens. The initial clear-screen for the first **refresh()** call is generated by initially setting `clear` to be TRUE for **curscr**. When this variable is set for the current screen (**curscr**), each **refresh()** generates a clear screen.

_leave

TRUE if the current (y, x) coordinates and the cursor are to be set to the character position following the last character changed on the terminal, or not moved if there is not a change.

_scroll

TRUE if scrolling is allowed.

_csbp

Current Standout Bit Pattern: The attribute pattern for characters that are written to the window in **standout** mode. See the `_STANDOUT` flag. See Figure 5-6 on page 5-28 for the patterns that can be combined into this variable.

****_y**

A pointer to an array of lines which describe the terminal. The expression:

```
_y[i]
```

is a pointer to the `i`th line, and:

```
_y [i][j]
```

is the `j`th character on the `i`th line.

****_a**

A pointer to the attribute array space. The expression:

```
_a [i][j]
```

is a pointer to the attribute variable (of data type ATTR) that corresponds to the `j`th character on the `i`th line, represented as `_y [i][j]` in the array specified by the ****_y** field.

struct _win_st *_view

A pointer to the original window from a viewport window.

PANEL Structure

The extended curses library routines use a structure, ***PANEL***, to hold information about each panel that it is working with. Figure B-2 shows the contents of that structure.

```
#define PANEL   struct Panel

struct  Panel
  {
  short int     p_depth ;
  short int     p_width ;
  short int     orow    ;
  short int     ocol    ;
  char          *title  ;
  char          divty   ;
  char          bordr   ;
  PANEL         *p_under;
  PANEL         *p_over ;
  PANE          *fpane  ;
  PANE          *dpane  ;
  PANE          *apane  ;
  WINDOW        *p_win  ;
  int           dfid    ;
  char          plobsc  ;
  char          plmodf  ;
  char          PLfill[6] ;
  }             ;
```

Figure B-2. Structure Definition for PANEL

The variables in this structure perform the following functions:

p_depth Number of rows in panel

p_width Number of columns in panel

orow Origin row (top left)

ocol Origin column

***title** Title string pointer

divty Divide type code

bordr Border flag byte

The following fields are used to relate multiple panels on the display:

***p_under** Next panel in chain under this panel

***p_over** Previous panel in chain over this panel

The following fields are used by the library routines. Do not change these fields directly:

***fpane** First pane after divisions

***dpane** First root pane for div

***apane** Current active pane

***p_win** Window struct for panel

dfid External panel ident

plobsc Panel obscured flag

plmodf Panel modified flag

PLfill[6] Not used

PANE Structure

The extended curses library routines use a structure, ***PANE***, to hold information about each pane that it is working with. Figure B-3 on page B-7 shows the contents of that structure.

```
#define PANE    struct Pane

struct  Pane
  {
  short int     w_depth ;
  short int     w_width ;
  short int     v_depth;
  short int     v_width ;
  short int     orow    ;
  short int     ocol    ;
  PANE          *vscr   ;
  PANE          *hscr   ;
  PANE          *nxtpn  ;
  PANE          *prvpn  ;
  PANE          *divs   ;
  PANE          *divd   ;
  char          divty   ;
  short int     divsz   ;
  char          divszu  ;
  char          bordr   ;
  WINDOW        *w_win  ;
  WINDOW        *v_win  ;
  int           pnvsid  ;
  PANEL         *hpanl  ;
  PANEPS        *exps   ;
  char          alloc   ;
  char          pnobsc ;
  char          pnmodf ;
  char          PNfill[6]
  }             ;
```

Figure B-3. Structure Definition for PANE

The variables in this structure perform the following functions:

w_depth Rows of data in presentation space for this pane.

w_width Columns of data in presentation space for this pane.

v_depth Rows being shown on the display of this pane including space for borders.

v_width Columns being shown on the display of this pane including space for borders.

orow Top row on panel of view for this pane (including the border).

ocol First column on panel of view for this pane (including the border).

***vscr** Pane to scroll vertically with this pane.

***hscr** Pane to scroll horizontally with this pane.

***nxtpn** Next pane in chain.

***prvpn** Previous pane in chain.

***divs** Next pane that is part of current division specification.

***divd** Start of division of this pane into smaller parts.

divty Division type code that applies to divisions of this pane. May have the following values:

- **Pdivtyv '0'** Divide vertical dimension of this pane. Divisions appear above each other.
- **Pdivtyh '1'** Divide horizontal dimension of this pane. Divisions appear beside each other.

divsz Division size specification:

divszu Division size unit specification that indicates the form for divsz value using one of the following values:

- **Pdivszc '1'** Size is a fixed constant. Fixed constants must be in the range from 1 to the dimension being divided.
- **Pdivszp '2'** Size is a proportional value. Proportional values must be in the range of 1 to 10,000. They represent the number of 10,000ths of the available screen space to assign to the pane.
- **Pdivszf '0'** Size is float. A **float** pane shares an equal amount of the available screen space with all other panes that have the float attribute.

bordr	Border flag for this pane.
***w_win**	Pointer to pspace window.
***v_win**	Pointer to view window.
pnvsid	External identifier for this p-pspace and view window.
***hpanl**	Pointer to panel that contains this pane.
***exps**	Pointer to chain of extra p-spaces for this pane.
alloc	An allocation flag that indicates whether ecdfpl allocated the window so that ecrlpl should free it.
pnobsc	A flag that indicates that the pane is obscured by an overlayed panel.
pnmodf	A flag that indicates the pane was modified.
PNfill[6]	Not used.

Appendix C. RT Printer Support Data Stream

Using Printers from A Program

The printer support of the RT system allows a program to produce output on any installed printer, as long as the program produces an output data stream that conforms to the control and data characters defined in this appendix. Printer support changes that data stream into the specific data stream that the installed printer needs.

Figure C-1 on page C-3 lists the printer control codes to use when printing using the RT printer support. If the printer can perform the function by itself, printer support passes the codes directly to the printer. If the codes do not work on the printer that is installed on your system, printer support performs one of the following actions:

- Tries to emulate the control with functions that the installed printer does have.
- Removes the control from the output stream. The function is not performed.

The three code columns in the table show different representations of the same code, depending on how you enter the code into the data stream:

Control Name This column shows the name of the control character. In many cases this name is the same as the keyboard keys that produce the required ASCII code for the control code.

Hex Code This column shows the hexadecimal representation of the control code.

ASCII Code This column shows the decimal representation of the control code.

Category	Function Performed	Control Name	Hex Code	ASCII Code
Control:	Provides a null value. Used as a list terminator.	NUL	00	0
	Sounds the buzzer. [2]	BEL	07	7
	Prints the next character as a printable character. The next character is a control with an ASCII value of less than 32.	ESC ^	1B5E	27 94
	Prints more than one character with an ASCII value that is below 32.	ESC \ *m n c*	1B5C *m n c*	27 92 *m n c*
	Clears the printer memory of all data waiting to be printed following the last received line feed. [1]	CAN	18	24
	Sets resolution for raster image print (*n* indicates a string of control bytes).[1]	ESC [O *n*	1B5B4F *n*	27 91 79 *n*
	Performs a printer power on reset.[1]	ESC [K	1B5B4B 0100 0	27 91 75 1 0 0
Positioning the Printhead:	Sets back space.	BS	08	8
	Sets horizontal tab.	HT	09	9
	Sets horizontal tabs (*n* is a list of one or more tab positions).	ESC D *n* NUL	1B44*n*00	27 68 *n* 0

Figure C-1 (Part 1 of 7). Printer Control Codes

1 Use with the Color Jet Printer.

2 Do not use these controls when using the print queue.

3 These controls may not work on the installed printer. Use *passthrough* mode to send these codes to the printer.

Category	Function Performed	Control Name	Hex Code	ASCII Code
	Sets tab stops to power-on settings.	ESC R	1B52	27 82
	Sets line feed.	LF	0A	10
	Sets reverse line feed.	ESC]	1B5D	27 93
	Starts automatic line feed.	ESC 5 1	1B35 1	27 53 1
	Stops automatic line feed.	ESC 5 0	1B35 0	27 53 0
	Provides a carriage return (no line feed).	CR	0D	13
	Provides a vertical tab.	VT	0B	11
	Sets vertical tabs (*n* is a list of tab positions).	ESC B *n* NUL	1B42 *n* NUL	27 66 *n* NUL
Paper Control:	Provides a form feed.	FF	0C	12
	Sets top of forms. [2]	ESC 4	1B34	27 52
	Ignores End of Forms. [2]	ESC 8	1B38	27 56
	Respects End of Forms. [2]	ESC 9	1B39	27 57
	Sets skip perforation [2] (*n* is lines to skip).	ESC N *n*	1B4E *n*	27 78 *n*
	Stops skip perforation [2]	ESC O	1B4F	27 79
Formatting the Page Image:	Uses 12 characters-per-inch printing.	ESC :	1B3A	27 58
	Sets 1/8″ Line spacing.	ESC 0	1B30	27 48
	Starts *n*/72″ Line spacing.	ESC 2	1B32	27 50
	Sets *n*/72″ Line spacing.	ESC A *n*	1B41 *n*	27 59 *n*

Figure C-1 (Part 2 of 7). Printer Control Codes

Category	Function Performed	Control Name	Hex Code	ASCII Code
	Sets Page Length. [2] (*n* is lines per page).	ESC C *n*	1B43 *n*	27 67 *n*
	Sets Page Length [2] (*n* is inches per page).	ESC C 0 *n*	1B43 0 *n*	27 67 0 *n*
	Sets left and right margins (*m* and *n* are column numbers).	ESC X *m n*	1B58 *m n*	27 88 *m n*
	Sets top and bottom margins (*m* and *n* are length of control; *t1 t0* are high/low-order bytes of top margin; *b1 b0* are high/low-order bytes of bottom margin).	ESC [S *m n t1 t0 b1 b0*	1B5B53 *m n t1 t0 b1 b0*	27 91 83 *m n t1 t0 b1 b0*
	Starts automatic line justification.	ESC M 1	1B4D 1	27 77 1
	Stops automatic line justification.	ESC M 0	1B4D 0	27 77 0
	Starts proportional spacing.	ESC P 1	1B50 1	27 80 1
	Stops proportional spacing.	ESC P 0	1B50 0	27 80 0
	Sets print resolution for draft quality font and ESC K image data (*n* indicates the resolution value).[1]	ESC [0 *n*	1B5B30 0100 *n*	27 91 48 1 0 *n*
	Sets presentation surface color (*n* indicates the available colors).[1]	ESC [L *n*	1B5B4C 0200 00 *n*	27 91 76 2 0 *n*
Controlling the Ribbon:	Sets color band 1 (yellow).	ESC y	1B79	27 121
	Sets color band 2 (magenta).	ESC m	1B6D	27 109
	Sets color band 3 (cyan).	ESC c	1B63	27 99

Figure C-1 (Part 3 of 7). Printer Control Codes

Category	Function Performed	Control Name	Hex Code	ASCII Code
	Sets color band 4 (black).	ESC b	1B62	27 98
	Sets automatic ribbon band shift.	ESC a	1B61	27 97
	Sets the background color for Text, ESC K, or ESC [0 printing (*n* indicates the available colors).[1]	ESC [N *n*	1B5B4E 0100 *n*	27 91 78 1 0 *n*
	Sets the foreground color for Text, ESC K, or ESC [0 printing (*n* indicates the available colors).[1]	ESC [M *n*	1B5B4D 0100 *n*	27 91 77 1 0 *n*
	Swaps the foreground and background printing color (*n* = 0 indicates normal printing; *n* = 1 indicates reversing background and foreground colors).[1]	ESC []	1B5B5D 0100 *n*	27 91 93 1 0 *n*
Selecting Print Mode:	Starts double wide printing.	SO	0E	14
	Stops double wide printing.	DC4	14	20
	Starts double wide continuous printing.	ESC W 1	1B57 1	27 87 1
	Stops double wide continuous printing.	ESC W 0	1B57 0	27 87 0
	Starts compressed printing.	SI	0F	15
	Stops compressed printing.	DC2	12	18
	Starts underline printing.	ESC -1	1B2D 1	27 45 1
	Stops underline printing.	ESC -0	1B2D 0	27 45 0
	Starts emphasized printing.	ESC E	1B45	27 69
	Stops emphasized printing.	ESC F	1B46	27 70

Figure C-1 (Part 4 of 7). Printer Control Codes

Category	Function Performed	Control Name	Hex Code	ASCII Code
	Starts double strike printing.	ESC G	1B47	27 71
	Stops double strike printing.	ESC H	1B48	27 72
	Starts superscript printing.	ESC S 0	1B53 0	27 83 0
	Starts subscript printing.	ESC S 1	1B53 1	27 83 1
	Stops superscript or subscript printing.	ESC T	1B54	27 84
	Starts color underline, bypassing white space (*n* defines the available color).[1]	ESC [B 1 *n*	1B5B42 0200 1 *n*	27 91 66 2 0 1 *n*
	Starts continuous color underline (*n* defines the available color).[1]	ESC [B 0 *n*	1B5B42 0200 0 *n*	27 91 66 2 0 0 *n*
	Stops underline.	ESC [E	1B5B450000	27 91 69 0 0
Selecting the Character Set:	Uses PC character set 2.	ESC 6	1B36	27 54
	Uses PC character set 1.	ESC 7	1B37	27 55
	Selects font (*n* specifies the font; varies with printer type).	ESC I *n*	1B49 *n*	27 73 *n*
	Sets graphic set ID (*c* selects graphic set 0, 1 or 2).	ESC [T 1 0 *c*	1B5B54 1 0 *c*	27 91 84 1 0 *c*
Using Bit Image Graphics: [3]	Sets bit graphics normal (*n* is a string of control bytes).	ESC K *n*	1B4B *n*	27 75 *n*
	Sets graphics dual-half speed (*n* is a string of control bytes).	ESC L *n*	1B4C *n*	27 76 *n*

Figure C-1 (Part 5 of 7). Printer Control Codes

Category	Function Performed	Control Name	Hex Code	ASCII Code
	Sets bit graphics dual-normal speed (*n* is a string of control bytes).	ESC Y *n*	1B59 *n*	27 89 *n*
	Sets bit graphics high-half speed (*n* is a string of control bytes).	ESC Z *n*	1B5A *n*	27 90 *n*
	Sets aspect ratio to 1:1.	ESC n 1	1B6E 1	27 110 1
	Sets aspect ratio to 5:6.	ESC n 0	1B6E 0	27 110 0
	Moves carriage to home position.	ESC <	1B3C	27 60
	Moves right *n*/120.	ESC d *n*	1B64 *n*	27 100 *n*
	Moves left *n*/120.	ESC e *n*	1B65 *n*	27 101 *n*
	Starts unidirectional printing.	ESC U 1	1B551	27 85 1
	Stops unidirectional printing.	ESC U 0	1B550	27 85 0
	Sets 7 dot line spacing.	ESC 1	1B31	27 49
	Sets graphics line spacing (*n* is the number of 1/216-inch steps).	ESC 3 *n*	1B33 *n*	27 51 n
	Sets variable space line feed (*n* is the number of 1/216-inch steps).	ESC J *n*	1B4A *n*	27 74 *n*
	Moves vertical presentation down the page in number of dots (*n* indicates how far to move the presentation). [1]	ESC [U *n*	1B5B55 0100 *n*	27 91 85 1 0 *n*
Selecting a Printer:	Selects the printer to accept data. [2]	DC1	11	17
	Deselects the printer so as to not receive data. [2]	DC3	13	19

Figure C-1 (Part 6 of 7). Printer Control Codes

Category	Function Performed	Control Name	Hex Code	ASCII Code
	Sets initialize function on. [2]	ESC ? 1	1B3F 1	27 63 1
	Sets initialize function off. [2]	ESC ? 0	1B3F 0	27 63 0
	Queries a parallel attached printer for identification. If the device queried is equal to *n*, this printer deactivates the select line. [2]	ESC Q *n*	1B51 *n*	27 81 *n*

Figure C-1 (Part 7 of 7). Printer Control Codes

1 Use with the Color Jet Printer.

2 Do not use these controls when using the print queue.

3 These controls may not work on the installed printer. Use *passthrough* mode to send these codes to the printer.

Appendix D. ASCII Characters

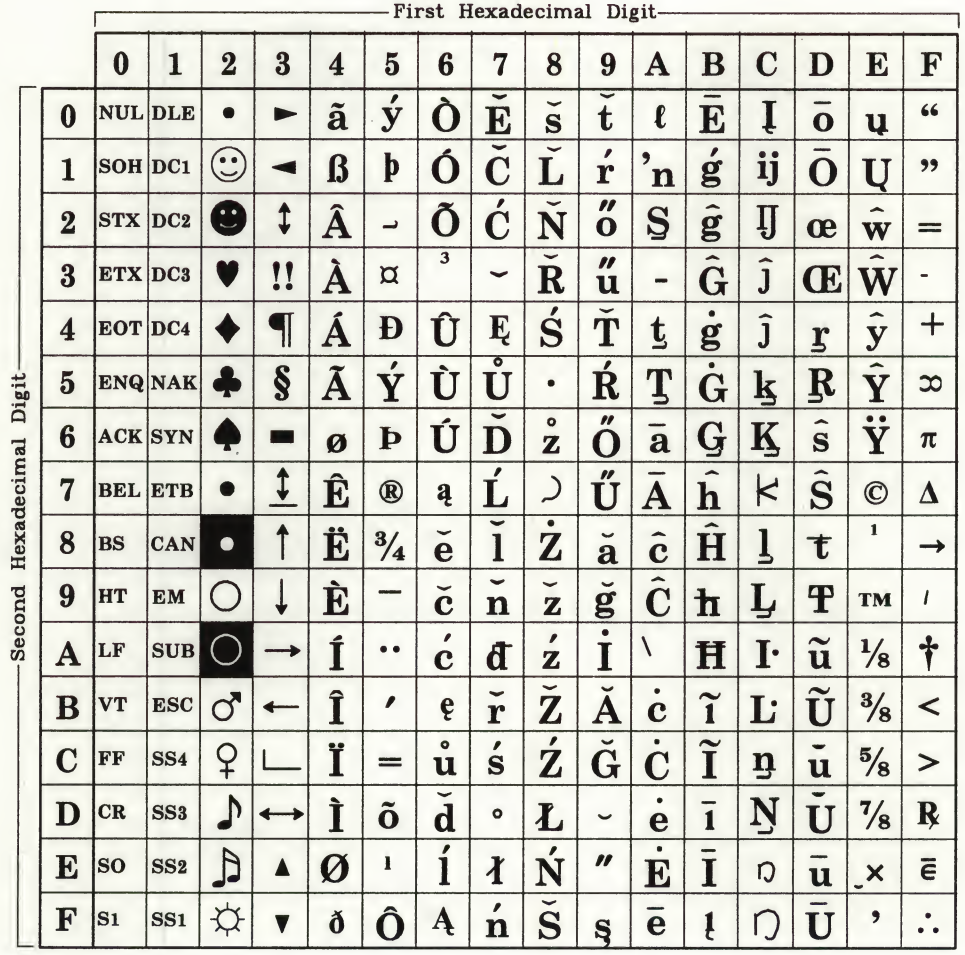
Japanese Language Support Information

Tables of Japanese language characters are in the publication N:GC18-2040-2, available from IBM Japan.

Second Hexadecimal Digit	First Hexadecimal Digit															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	BLANK (SPACE)	0	@	P	‘	p	Ç	É	á	⁞	└	ð	Ó	-
1	SOH	DC1	!	1	A	Q	a	q	ü	æ	í	⁞	┴	Ð	ß	±
2	STX	DC2	″	2	B	R	b	r	é	Æ	ó	⁞	┬	Ê	Ô	=
3	ETX	DC3	#	3	C	S	c	s	â	ô	ú	\|	├	Ë	Ò	¾
4	EOT	DC4	$	4	D	T	d	t	ä	ö	ñ	┤	─	È	õ	¶
5	ENQ	NAK	%	5	E	U	e	u	à	ò	Ñ	Á	┼	ı	Õ	§
6	ACK	SYN	&	6	F	V	f	v	å	û	a	Â	ã	Í	µ	÷
7	BEL	ETB	’	7	G	W	g	w	ç	ù	o	À	Ã	Î	þ	¯
8	BS	CAN	(	8	H	X	h	x	ê	ÿ	¿	©	╚	Ï	Þ	°
9	HT	EM	)	9	I	Y	i	y	ë	Ö	®	╣	╔	┘	Ú	··
A	LF	SUB	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Û	•
B	VT	ESC	+	;	K	[	k	{	ï	ø	½	╗	╦	■	Ù	1
C	FF	SS4	,	<	L	\	l	\|	î	£	¼	╝	╠	▄	ý	3
D	CR	SS3	—	=	M	]	m	}	ì	Ø	¡	¢	═	¦	Ý	2
E	SO	SS2	.	>	N	^	n	~	Ä	x	«	¥	╬	Ì	-	▮
F	S1	SS1	/	?	O	_	o	Δ	Å	ƒ	»	┐	¤	▀	’	BLANK 'FF'

A5AC6006

Figure D-1. Code Page 0



Second Hexadecimal Digit	First Hexadecimal Digit															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	DLE	•	►	ã	ý	Ò	Ě	š	ť	ℓ	Ē	Į	ō	ų	“
1	SOH	DC1	☺	◄	ß	þ	Ó	Č	Ľ	ŕ	’n	ģ	ij	Ō	Ų	”
2	STX	DC2	☻	↕	Â	-	Õ	Ć	Ň	ő	Ş	ĝ	Ĳ	œ	ŵ	=
3	ETX	DC3	♥	!!	À	¤	³	˘	Ř	ű	-	Ĝ	ĵ	Œ	Ŵ	-
4	EOT	DC4	♦	¶	Á	Đ	Û	Ę	Ś	Ť	ţ	ġ	ĵ	ŗ	ŷ	+
5	ENQ	NAK	♣	§	Ã	Ý	Ù	Ů	·	Ŕ	Ţ	Ġ	ķ	Ŗ	Ŷ	∞
6	ACK	SYN	♠	▬	ø	Þ	Ú	Ď	ž	Ő	ā	Ģ	Ķ	ŝ	Ÿ	π
7	BEL	ETB	•	↨	Ê	®	ą	Ĺ	ˇ	Ű	Ā	ĥ	ĸ	Ŝ	©	Δ
8	BS	CAN	◘	↑	Ë	¾	ě	ľ	Ż	ă	ĉ	Ĥ	ļ	ŧ	¹	→
9	HT	EM	○	↓	È	¯	č	ň	ž	ğ	Ĉ	ħ	Ļ	Ŧ	™	ı
A	LF	SUB	◙	→	Í	··	ć	đ	ź	İ	\	Ħ	I·	ũ	⅛	†
B	VT	ESC	♂	←	Î	’	ę	ř	Ž	Ă	ċ	ĩ	Ŀ	Ũ	⅜	<
C	FF	SS4	♀	∟	Ï	=	ů	ś	Ź	Ğ	Ċ	Ĩ	ŋ	ŭ	⅝	>
D	CR	SS3	♪	↔	Ì	õ	ď	°	Ł	˘	ė	ī	Ŋ	Ŭ	⅞	Ŗ
E	SO	SS2	♫	▲	Ø	ı	Í	ł	Ń	″	Ė	Ī	ŋ	ū	ˍ×	ē
F	S1	SS1	☼	▼	ð	Ô	Ą	ń	Š	ş	ē	ŀ	∩	Ū	’	∴

A5AC6007

Figure D-2. Code Page 1

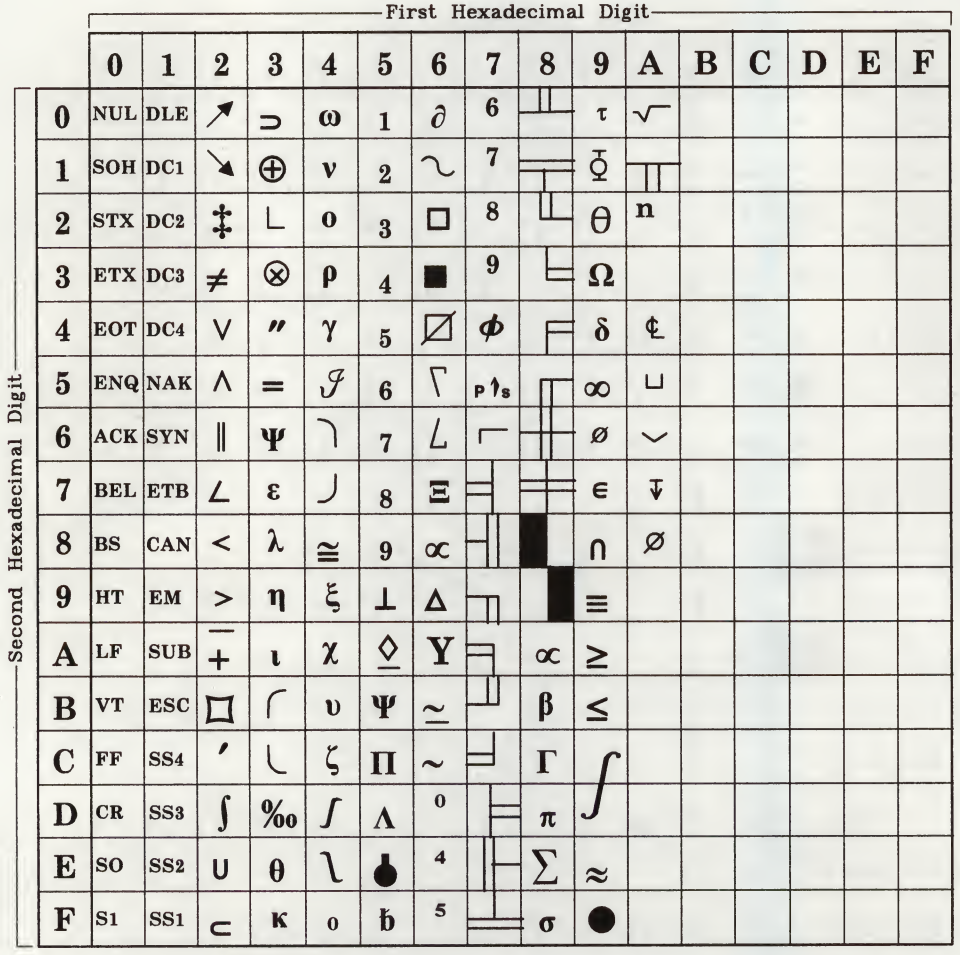


Figure D-3. Code Page 2

Appendix E. Porting 4.3BSD Programs

This appendix provides information to help application programmers port 4.3BSD (unix version 4.3 Berkeley System Distributions) programs to AIX. It also provides a lists of 4.3BSD routines and commands supported in AIX. These routines are found in the libc.a.

Note: The **copysign** procedure is an exception. This routine is found in **libm.a**.

You should read this entire appendix before:

- Porting existing programs
- Writing new AIX programs that use BSD functions.

Once you read this appendix refer back to specific discussions or to individual routine descriptions as needed.

BSD Library Subroutines

The following is a list of 4.3BSD subroutines available in AIX:

accept
alphasort
bcopy
bind
bstring: bcmp, bzero, ffs
connect
copysign
directory: closedir, opendir, readdir, rewinddir, telldir, seekdir
dup2
fsync
ftruncate
getdtablesize
getgroups
gethostbyaddr, gethostbyname, sethostent, endhostent
gethostid, sethostid
gethostname, sethostname
getitimer, setitimer
getnetent, getnetbyaddr, getnetbyname, setnetent, endnetent
getpeername
getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent
getservent, getservbyname, getservbyport, setservent, endservent
getsockname
getsockopt, setsockopt
gettimeofday
getwd
htonl, htons, ntohl, ntohs
index, rindex
inet_addr, inet_network, inet_ntoa, inet_makeaddr, inet_lnaof, inet_netof
initgroups
insque, remque
killpg
listen
mkdir
random, srandom, initstate, setstate
rcmd, rresvport, ruserok
readv
reboot
recv, recvfrom, recvmsg
rename
resolver: res_mkquery, res_send, res_init, dn_comp, dn_expand, getshort, getlong, putshort, putlong
rexec
rmdir
scandir
select
send, sendto, sendmsg
setbuffer, setlinebuf
setgroups
seteuid
setreuid
shutdown
sigblock
sigpause
sigsetmask
sigstack
sigvec
socket
socketpair
syslog, openlog, closelog, setlogmask
utimes
varargs
wait3
writev

BSD to AIX Compatible Subroutines

AIX provides some BSD-compatible functions to limit changes to source code when porting. These functions are part of the libc.a. They link automatically with a program to provide compatible operation. In most cases, these functions simply invoke the equivalent AIX function.

The following functions carry restrictions when porting into AIX

gettimeofday This function is limited to 1/60 of a second.

getitimer, setitimer In AIX these functions use the socket device driver to get to the kernel. Timer resolution is also limited to 1/60 of a second.

Notes:

1. The driver may not be able to open the device if your program is out of descriptors.
2. These functions only support the **ITIMER_REAL** timer.

killpg This function is implemented using the AIX function **kill** with the "pip" (pgrp) negated. The **killpg** function does not support sending signals to members of process group 1. In this case, all processes except 0 and 1 will get the signal.

setbuffer, setlinebuf The **setbuffer** can be used to setup **stdio** stream buffering after opening but before the stream is read from or written to. The **setlinebuf** can be used to setup **stdio** stream buffering at any time. These are implemented using System V **setbuf**.

seteuid, setreuid This function uses the socket device driver to get to the kernel. It may not be able to open a device if you are out of descriptors. Except for this it operates in the same manner as BSD.

If you need the **setreuid** or **seteuid** functions construct them as follows:

```
#define seteuid(id) setreuid(-1,(id)
#define setruid(id) setreuid((id),-1)
```

syslog Since AIX sockets do not create file names for **AF_UNIX** sockets, there is no **/dev/log** file created when **syslog** starts up.

utimes This function is limited to one second resolution.

Notes:

1. **WUNTRACED** is not supported in AIX, but **WNOHANG** is.
2. The **rusage** parameter is not supported. The information it provides is not available in any way.
3. The return status is limited to what the System V **wait** returns.

For more information on BSD-compatible functions see the discussions of individual routines in *AIX Operating System Technical Reference.*

BSD to AIX Subroutines with Differences

Some 4.3BSD library routines function differently in AIX. This section provides information on working with these differences.

strcpyn Replace calls to this subroutine with the **strncpy** subroutine.

strcatn Replace calls to this subroutine with the **strncat** subroutine.

_doprnt The **_doprnt** subroutine carries out internal **printf** functions. In AIX, it returns the count of characters printed or **EOF**.

_doprnt(*format*, *args*, *stream***)**
char *format*;
va_list *args*;
FILE *stream*;

sleep The **sleep** subroutine uses the **alarm**, **pause**, and **signal** system calls to achieve a simple delay. Since signal handling in 4.3BSD and AIX is not the same, 4.3BSD code using **SIGALRM** from **setitimer** may not work. This is particularly true if the **sleep** call is issued inside a signal handler, such as when exiting.

To solve this problem, replace the **sleep** subroutine with a spin loop or a private version of sleep made with the **setitimer** subroutine and the **signal** system call. Also, define **BSD_REMAP_SIGNAL_TO_SIGVEC** so that 4.3BSD-compatible signals are used.

Porting 4.3BSD Applications to AIX

This section provides programmers with information necessary to use the 4.3BSD functions provided by AIX. It explains some useful techniques you may need when porting 4.3BSD programs to AIX.

When porting, use **makefile** changes whenever possible. You should change the original source code only when absolutely necessary. Changes to the **makefile** include:

- Add **-DBSD_INCLUDES** to your compile command to give you access to 4.3BSD include files.
- Add **-DBSD_REMAP_SIGNAL_TO_SIGVEC** to your compile command if you use the **signal** system call as it is used in 4.3BSD.

To modify source code for the **cpp** C language preprocessor, enclose your changes in **ifdef** statements similar to the following:

```
#ifdef AIX
        <code for AIX specific version>
#else
        <code for 4.3BSD specific version>
#endif
```

Note: The define for the AIX Operating System is **AIX**.

You should be aware when porting 4.3BSD application programs to AIX, system calls in AIX are not restarted after receiving a signal as in 4.3BSD. Instead, AIX system calls return a value of -1 and set the global variable **errno** to **EINTR** when interrupted by a signal.

4.3BSD Include Files Provided by AIX

AIX does not provide all the 4.3BSD include files, available files are only those commonly required for porting applications. This section contains a list of include files provided by AIX and instructions on how to access them.

The following includes are provided by AIX:

General Include Files

- **fcntl.h**
- **sgtty.h**
- **strings.h**
- **sysexits.h**
- **sys/dir.h**
- **sys/file.h**
- **sys/ioctl.h**
- **sys/msgbuf.h**
- **sys/param.h**
- **sys/resource.h**
- **sys/signal.h**
- **sys/syslog.h**
- **sys/time.h**
- **sys/ttychars.h**
- **sys/ttydev.h**
- **sys/types.h**
- **sys/wait.h**
- **math.h**
- **sys/socket.h**
- **sys/uio.h**

Network Include Files

- **arpa/ftp.h**
- **arpa/inet.h**
- **arpa/nameser.h**
- **arpa/telnet.h**
- **arpa/tftp.h**
- **net/af.h**
- **net/if.h**
- **net/if_arp.h**
- **netinet/in.h**
- **netinet/in_systm.h**
- **netinet/in_var.h**
- **netdb.h**
- **resolv.h**
- **sys/un.h**

Include File Restrictions

Before using the **sys/dir.h** or **sys/signal.h** files, you should be aware of the following:

dir.h This file defines and declares 4.3BSD-type directory entry structures. Use it *only* if you are using the **opendir**, **closedir**, **readdir**, **rewinddir**, **seekdir**, and **telldir** functions.

Note: The **dir.h** file causes problems if used with the AIX **user.h** file or other files that expect an AIX-type directory structure.

signal.h This include file is used primarily to remap signal names from 4.3BSD names to AIX names. In addition, you can use **signal.h** to change the definition of the **signal** system call. The AIX **signal** system call handles signals differently than 4.3BSD. To make the system calls compatible, the **sys/signal.h** file contains a static function **signal()**. This function is surrounded by **#ifdef BSD_REMAP_SIGNAL_TO_SIGVEC** statements.

Define **BSD_REMAP_SIGNAL_TO_SIGVEC** if you have a pure BSD application. This specifies signals are not reset after being caught. Do *not* define this symbol if you have a mixed 4.3BSD and AIX application that expects to reset signals in the signal handler.

Notes:

1. The typedefs in the 4.3BSD networking include files used by the sockets routines, have changed from **u_long** to **ulong**.
2. To avoid duplicate defines, the 4.3BSD include files contain the text of any applicable AIX include files. This is true whether they are for the same name or for related purposes.
3. You can use 4.3BSD include files that use the convention of defining **_H_***filename***.h** multiple times without causing errors.
4. **#ifdef BSD_COMPILE_ONLY** statements enclose the parts of 4.3BSD include files not relevant to AIX. If you only want to compile, you need to define this symbol. Names and symbols from these files which are not defined indicate your program will not work in AIX without changes to the source code.

Compiling 4.3BSD Include Files

To compile a 4.3BSD source program that uses these include files, use one of two methods:

1. Add the **-DSBD_INCLUDES** option to the **cc** command in the **makefile** or on the command line.

 You would use this method with simple programs that do not contain much kernel-specific code. It does not require changes to the program source code and is the preferred method.
2. Explicitly define and undefine **BSD_INCLUDES** as needed in the program source. You would use this for complex programs, and *only* as a last resort.

4.3BSD tty Devices

Since the AIX and 4.3BSD drivers are different, you may need to change your source code's use of the **tty** driver. If your 4.3BSD program uses the **curses** library, **libcurses.a**, **tty** issues should be handled at that level. If your 4.3BSD program sets **tty** modes, it is recommended you insert **ifdef** statements around the 4.3BSD **tty** source code and replace it with new source code using the AIX **termio** structures.

Note: AIX uses the **terminfo** subroutine instead of the **termcap** subroutine used in 4.3BSD.

When writing or updating code that gets and uses **ptys**, keep in mind:

- AIX and 4.3BSD use different naming conventions for **ptys**.

In 4.3BSD, the YP slave server side is named **/dev/tty***xy*, while the YP master server side is named **/dev/pty***xy*. In these names, *x* represents **p** or **q** or **r**. The *y* represents any one value from **0** to **9** or from **a** to **f**. The only difference in the names is that the YP master server is called **/dev/tty**, while the YP slave server is called **/dev/pty**.

In AIX, the YP master server side is named **/dev/ptc***n*, and the YP slave server side is named **/dev/pts***n*, with *n* in both cases representing a decimal number greater than or equal to 0. The difference in names is that the YP master server is called **/dev/ptc**, while the YP slave server is called **/dev/pts**.

To assist in porting, the appropriate BSD names link to the appropriate AIX names. For example, `/dev/ptyp0` links to `/dev/ptc0` and `/dev/ttyp0` links `/dev/pts0`.

- **ptys** are obtained differently in AIX than 4.3BSD.

 In AIX, a **getty** command can be running on the YP slave server side of a **pty**, waiting for the **telnet** program or a similar program to attach itself on the YP master server side. This possibility prevents programs from trying YP master server **ptys** until one opens, then assuming the YP slave server side is available, as happens in 4.3BSD.

 Programs that need to get **ptys** in AIX must check the **/etc/portstatus** file to see which **tty** ports, including **ptys**, are enabled. For more information, see "portstatus" in *AIX Operating System Technical Reference.*

4.3BSD ioctls

The AIX and 4.3BSD operating systems do not support the same **ioctl**s. For a list of the 4.3BSD **ioctl**s AIX does not support, see the sections of the **/usr/include/sys/ioctl.h** file between the following lines:

#ifdef BSD_COMPILE_ONLY
#endif BSD_COMPILE_ONLY

Several of the 4.3BSD **ioctl**s have equivalent **fcntl**s, but AIX has only the **fcntl** calls. In addition, some 4.3BSD **fcntl**s are not available in AIX, so **ioctl**s must be used instead. For example:

Used in 4.3BSD	**Used in AIX**
ioctl **FIONBIO**	fcntl **F_SETFL** to set **NDELAY**
fcntl **F_SETOWN**	ioctl **FIOSETOWN** to set owner (sockets only)

These differences can be handled by using the applicable **ioctl** or **fcntl** in place of the original code.

BSD Commands Available in AIX

The following is a list of BSD commands available in AIX, and the program in which you will find them.

Command	Program
accton	Multi-User Services
adb	Extended Services
admin	Extended Services
basename	AIX Operating System
bellmail	AIX Operating System
cal	Extended Services
cb	AIX Operating System
clri	AIX Operating System
cmp	AIX Operating System
comb	Extended Services
comm	Extended Services
config	AIX Operating System
cu	Extended Services
dc	Extended Services
delta	Extended Services
diffh	Extended Services
dircmp	Extended Services
fingerd	TCP/IP
fmt	AIX Operating System
ftp	TCP/IP
ftpd	TCP/IP
get	Extended Services
graph	Multi-User Services
inetd	TCP/IP
lorder	AIX Operating System
mail, Mail	AIX Operating System
mknod	AIX, Installation/Maintenance
named	TCP/IP
ncheck	AIX Operating System
netstat	TCP/IP
newform	Extended Services
nice	AIX Operating System
nohup	AIX Operating System
pr	AIX Operating System
ptx	Extended Services
pwd	AIX Operating System
rcp	TCP/IP
rexecd	TCP/IP
rlogin	TCP/IP
rlogind	TCP/IP
rmail	AIX Operating System
rmdel	Extended Services
rshd	TCP/IP
ruptime	TCP/IP
rwho	TCP/IP
rwhod	TCP/IP
sccsdiff	Extended Services
size	AIX Operating System
sleep	AIX Operating System
spline	Multi-User Services
split	AIX Operating System
strip	AIX Operating System
sync	AIX Installation/Maintenance
talkd	TCP/IP
tbl	Extended Services
tee	AIX Operating System
telnetd	TCP/IP
tftpd	TCP/IP
tsort	AIX Operating System
tty	AIX Operating System
unmount	AIX, Installation/Maintenance
uniq	AIX Operating System
units	Extended Services
uucico	Extended Services
uucp	Extended Services
uucpd	Extended Services
uulog	Extended Services
uuname	Extended Services
uux	Extended Services
uuxqt	Extended Services
wall	AIX Operating System
wc	AIX Operating System
what	AIX Operating System
whatnow	Extended Services
who	AIX Operating System
whois	TCP/IP

Command	Program
write	AIX Operating System
yacc	Extended Services

Note: AIX may contain other commands which are similar to BSD. However, these commands may function differently.

Additional Information

See the description of individual subroutines in *AIX Operating System Technical Reference.*

See the descriptions of individual BSD commands in *AIX Operating System Commands Reference.*

See the descriptions of individual BSD commands in *Interface Program for use with TCP/IP.*

Glossary

access. To obtain data from or put data in storage.

access permission. A group of designations that determine who can access a particular AIX file and how the user may access the file.

action. In **awk**, **lex** and **yacc**, a C language program fragment that defines what the program does when it finds input that it recognizes.

All Points Addressable (APA) display. A display that allows each pel to be individually addressed. An APA display allows for images to be displayed that are not made up of images predefined in character boxes. Contrast with *character display*.

allocate. To assign a resource, such as a disk file or a diskette file, to perform a specific task.

alphabetic. Pertaining to a set of letters a through z.

alphanumeric character. Consisting of letters, numbers and often other symbols, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

American National Standards Institute. An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards.

application. A program or group of programs that apply to a particular business area, such as the Inventory Control or the Accounts Receivable application.

application program. A program used to perform an application or part of an application.

ASCII. See *American National Standard Code for Information Interchange*.

attribute. A characteristic. For example, the attribute for a displayed field could be blinking.

auto carrier return. The system function that places carrier returns automatically within the text and on the display. This is accomplished by moving whole words that exceed the line end zone to the next line.

background process. (1) An activity that does not require operator intervention that can be run by the computer while the work station is used to do other work. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.

backup copy. A copy, usually of a file or group of files, that is kept in case the original file or files are unintentionally changed or destroyed.

backup diskette. A diskette containing information copied from a fixed disk or from another diskette. It is used in case the original information becomes unusable.

bad block. A portion of a disk that can never be used reliably.

base address. The beginning address for resolving symbolic references to locations in storage.

basename. The last element to the right of a full path name. A file name specified without its parent directories.

batch printing. Queueing one or more documents to print as a separate job. The operator can type or revise additional documents at the same time. This is a background process.

batch processing. A processing method in which a program or programs process records with little or no operator action. This is a background process. Contrast with *interactive processing*.

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

bit. Either of the binary digits 0 or 1 used in computers to store information. See also *byte*.

block. (1) A group of records that is recorded or processed as a unit. Same as *physical record*. (2) In data communications, a group of records that is recorded, processed, or sent as a unit. (3) A block is 512 bytes long.

block file. A file listing the usage of blocks on a disk.

block special file. A special file that provides access to a device which is capable of supporting a file system.

branch. In a computer program an instruction that selects one of two or more alternative sets of instructions. A conditional branch occurs only when a specified condition is met.

breakpoint. A place in a computer program, usually specified by an instruction, where execution may be interrupted by external intervention or by a monitor program.

buffer. (1) A temporary storage unit, especially one that accepts information at one rate and delivers it at another rate. (2) An area of storage, temporarily reserved for performing input or output, into which data is read, or from which data is written.

bug. A problem in the logic of a program that causes the program to perform differently than expected.

byte. The amount of storage required to represent one character; a byte is 8 bits.

C language. A general-purpose programming language that is the primary language of the AIX Operating System.

call. To activate a program or procedure at its entry point. Compare with *load*.

cancel. To end a task before it is completed.

carrier return. (1) In text data, the action causing line ending formatting to be performed at the current cursor location followed by a line advance of the cursor. Equivalent to the carriage return of a typewriter. (2) A keystroke generally indicating the end of a command line.

channel. One of 32 bits in a table used to represent which event classes are active or inactive. The most significant bit is called **channel 0** and the least significant bit is called **channel 31**.

character. A letter, digit, or other symbol.

character class. Ranges of characters that match a single character.

character display. A display that uses a character generator to display predefined character boxes of images (characters) on the screen. This kind of display can not address the screen any less than one character box at a time. Contrast with *All Points Addressable display*.

character key. A keyboard key that allows the user to enter the character shown on the key. Compare with *function keys.*

character position. On a display, each location that a character or symbol can occupy.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character special file. A special file that provides access to an input or output device. The character interface is used for devices that cannot or do not want to use a file system.

character string. A sequence of consecutive characters.

character variable. The name of a character data item whose value may be assigned or changed while the program is running.

child. (1) Pertaining to a secured resource, either a file or library, that uses the user list of a parent resource. A child resource can have only one parent resource. (2) In the AIX Operating System, child is a *process* spawned by a parent process that shares the resources of the parent process. Contrast with *parent.*

client. Computer or process that accesses the data, services, or resources of another computer or process on a network.

close. To end an activity and remove that window from the display.

code. (1) Instructions for the computer. (2) To write instructions for the computer; to *program.* (3) A representation of a condition, such as an error code.

code page. Arrays of code points (see *code point*) representing characters that establish ordinal sequence (numeric order) of characters. AIX uses 256-character code pages. Code page P0 consists of 1-byte characters that represent the ASCII, ISO, and EBCDIC character sets and additional characters and symbols. Lower code page P0 (0-127 ordinal) is the ASCII character set. Additional code pages consist of code points for 2-byte character representations. See *extended character.* Appendix D in this book contains tables of code page character sets.

code point. A 1- or 2-byte representation of a character. A byte can contain a single-shift value that indicates the second byte is a part of the same code point. In AIX, the single-shift byte indicates the code page of the character. Also in AIX, the second byte (only byte in the case of a 1-byte character) places the character in the code page array.

code segment. See *segment.*

collating sequence. The sequence in which characters are ordered within the computer for sorting, combining, or comparing.

collation. The process of character and string sorting based on alphabetical order, and, in AIX, on equivalence class.

collation table. Provides an ordered character set and character equivalence classes used by functions.

color display. A display device capable of displaying more than two colors and the shades produced via the two colors, as opposed to a monochrome display.

column. A vertical arrangement of text or numbers.

column headings. Text appearing near the top of columns of data for the purpose of identifying or titling.

command. A request to perform an operation or execute a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command line editing keys. Keys for editing the command line.

compile. (1) To translate a program written in a high-level programming language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

compiler. A program that reads program text from a file and changes the programming language statements in that file to a form that the system can understand.

component dump table. A structure used by the operating system VRM components to identify data structures that should be collected by the operating system VRM dump program.

compress. (1) To move files and libraries together on disk to create one continuous area of unused space. (2) In data communications, to delete a series of duplicate characters in a character string.

compression. A technique for removing strings of duplicate characters and for removing trailing blanks before transmitting data.

concatenate. (1) To link together. (2) To join two character strings.

concurrent groups. The ability to access files from many groups at the same time.

condition. An expression in a program or procedure that can be evaluated to a value of either true or false when the program or procedure is running.

configuration. The group of machines, devices, and programs that make up a computer system. See also *system customization.*

constant. A data item with a value that does not change. Contrast with *variable*

control block. A storage area used by a program to hold control information.

control program. Part of the AIX Operating System that determines the order in which basic functions should be performed.

controlled cancel. The system action that ends the job step being run, and saves any new data already created. The job that is running can continue with the next job step.

copy. The action by which the user makes a whole or partial duplicate of already existing data.

crash. An unexpected interruption of computer service, usually due to a serious hardware or software malfunction.

creation date. The program date at the time a file is created.

current directory. The currently active directory, displayed with the **pwd** command.

current file. In **make**, the file that the **make** command is working with at a given moment. **make** replaces the macro $* with the name of the current file.

current line. The line on which the cursor is located.

current screen. In **extended curses**, the actual image that is currently on the terminal.

current working directory. See *current directory.*

cursor. (1) A movable symbol (such as an underline) on a display, used to indicate to the operator where the next typed character will be placed or where the next action will be directed. (2) A marker that indicates the current data access location within a file.

cursor movement keys. The directional keys used to move the cursor.

customize. To describe (to the system) the devices, programs, users, and user defaults for a particular data processing system.

cylinder. All fixed disk or diskette tracks that can be read or written without moving the disk drive or diskette drive read/write mechanism.

daemon. See *daemon process.*

daemon process. A process begun by the root or the root shell that can be stopped only by the root. Daemon processes generally provide services that must be available at all times such as sending data to a printer.

data communications. The transmission of data between computers, or remote devices or both (usually over long distance).

data stream. All information (data and control information) transmitted over a data link.

declaration. A statement in a program that defines how a label is used.

default. A value, attribute, or option that is used when no alternative is specified by the operator.

default directory. The directory name supplied by the operating system if none is specified.

default drive. The drive name supplied by the operating system if none is specified.

default value. A value stored in the system that is used when no other value is specified.

delete. To remove. For example, to delete a file.

dependent work station. A work station having little or no stand-alone capability, that must be connected to a host or server in order to provide any meaningful capability to the user.

device. An electrical or electronic machine that is designed for a specific purpose and that attaches to your computer, for example, a printer, plotter, or disk drive.

device driver. A collection of routines that control the interface between I/O device adapters and the operating system. The primary components of an RT device driver are the device head, device handler, and device manager.

device name. A name reserved by the system that refers to a specific device.

diagnostic. Pertaining to the detection and isolation of an error.

diagnostic aid. A tool (procedure, program, reference manual) used to detect and isolate a device or program malfunction or error.

diagnostic routine. A computer program that recognizes, locates, and explains either a fault in equipment or a mistake in a computer program.

digit. Any of the numerals from 0 through 9.

directory. A type of file containing the names and controlling information for other files or other directories.

diskette. A thin, flexible magnetic plate that is permanently sealed in a protective cover. It can be used to store information copies from the disk or another diskette.

diskette drive. The mechanism used to read and write information on diskettes.

display device. An output unit that gives a visual representation of data.

display screen. The part of the display device that displays information visually.

display station. A device that includes a keyboard from which an operator can send information to the system and a display screen on which an operator can see the information sent to or received from the computer.

dump. (1) To copy the contents of all or part of storage, usually to an output device.
(2) Data that has been dumped.

dump data. The data collected by the VRM dump program. It is obtained from memory locations used by VRM components.

dump table entry. A record in the master dump table that identifies the location of a component dump table. All VRM components that need to have special data collected by the operating system VRM dump program need to generate a dump table entry.

EBCDIC. See *extended binary-coded decimal interchange code.*

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC set.

edit. To modify the form or format of data.

editor. A program used to enter and modify programs, text, and other types of documents.

emulation. Imitation; for example, when one computer imitates the characteristics of another computer.

enable. To make functional.

enter. To send information to the computer by pressing the **Enter** key.

entry. A single input operation on a work station.

environment. The settings for shell variables and paths set when the user logs in. These variables can be modified later by the user.

equivalence class. In AIX, a grouping of characters (or character strings) that are considered equal when processing the endpoints of a character range in regular expressions. For example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. Japanese Language Support does not support equivalence classes.

error-correct backspace. An editing key that performs editing based on a cursor position; the cursor is moved one position toward the beginning of the line, the character at the new cursor location is deleted, and all characters following the cursor are moved one position toward the beginning of the line (to fill the vacancy left by the deleted element).

error entry. A data structure containing a header of identifying information plus several bytes of defined data. Error entries are generated by error points and written to an error log file.

error ID. This is part of the data required by an error entry. It is a unique combination of three hexadecimal digits that identifies the component that generated the error entry.

error identifier. A three-character code used to identify error templates and to specify which error entries the error formatter should process. This code is based on the error ID; however, it uses alphanumeric characters instead of hexadecimal digits.

error point. A group of code statements that generates an error entry from within a software program. Error entries are generated when a software or hardware component encounters an error.

error type. One of six categories of errors. The type of an error is determined by the software program that generates the error. When you format an error log, you can specify which types of errors you want to format.

escape character. The backslash character, used to indicate to the shell that the next character is not intended to have the special meaning normally assigned to it by the shell.

event class. A number assigned to a group of trace points that relate to a specific subject or system component. The defined event classes are listed in the trace profile.

exit value. (1) A code sent to either standard output or standard error on completion of the command. (2) A numeric value that a command returns to indicate whether it completed successfully. Some commands return exit

values that give other information, such as whether a file exists. Shell programs can test exit values to control branching and looping.

expression. A representation of a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

extended character. A character other than a 7-bit ASCII character. An extended character can be a 1-byte code point with the 8th bit set (ordinal 128-255) or a 2-byte code point (ordinal 256 and greater).

fake target name. A control name used in a **makefile** that looks like a target name, but actually tells **make** to perform some operation differently.

feature. A programming or hardware option, usually available at an extra cost.

field. (1) An area in a record or panel used to contain a particular category of data. The smallest component of a record that can be referred to by a name. (2) In **extended curses**, an area in a presentation space where the program can accept operator input.

FIFO. See *first-in-first-out.*

file. A collection of related data that is stored and retrieved by an assigned name.

file descriptor. A small positive integer that the system uses instead of the file name to identify the file.

file name. The name used by a program to identify a file. See also *label.*

file specification (filespec). The name and location of a file. A file specification consists of a drive specifier, a path name, and a file name.

file system. The collection of files and file management structures on a physical or logical mass storage device, such as a diskette or minidisk.

filename. In DOS, that portion of the file name that precedes the extension.

filter. A command that reads standard input data, modifies the data, and sends it to standard output.

first-in-first-out (FIFO). A named permanent pipe. A FIFO allows two unrelated processes to exchange information using a pipe connection.

first level interrupt handler (FLIH). A routine that receives control of the system as a result of a hardware interrupt. One FLIH is assigned to each of the six interrupt levels.

fixed disk. A flat, circular, nonremoveable plate with a magnetizable surface layer on which data can be stored by magnetic recording.

fixed-disk drive. The mechanism used to read and write information on fixed disk.

flag. A modifier that appears on a command line with the command name that defines the action of the command. Flags in the AIX Operating System almost always are preceded by a dash.

flattened character. In AIX, an ASCII character created by translating an extended character to its ASCII equivalent in appearance. Code point information is lost; the character cannot be retranslated to an extended character.

font. A family or assortment of characters of a given size and style.

foreground. A mode of program execution in which the shell waits for the program specified on the command line to complete before returning your prompt.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) The pattern which determines how data is recorded.

formatted diskette. A diskette on which control information for a particular computer system has been written but which may or may not contain any data.

free list. A list of available space on each file system. This is sometimes called the free-block list.

full path name. The name of any directory or file expressed as a string of directories and files beginning with the root directory.

function. A synonym for procedure. The C language treats a function as a data type that contains executable code and returns a single value to the calling function.

function keys. Keys that request actions but do not display or print characters. Included are the keys that normally produce a printed character, but when used with the code key produce a function instead. Compare with *character key*.

generation. For some remote systems, the translation of configuration information into machine language.

Gid. See *group number*.

global. Pertains to information available to more than one program or subroutine.

global action. An action having general applicability, independent of the context established by any task.

global character. The special characters * and ? that can be used in a file specification to match one or more characters. For example, placing a ? in a file specification means any character can be in that position.

global search. The process of having the system look through a document for specific characters, words, or groups of characters.

global symbol. A symbol defined in one program module, but used in other independently assembled program modules.

graphic character. A character that can be displayed or printed.

group name. A name that uniquely identifies a group to the system.

group number (Gid). A unique number assigned to a group of related users. The group number can often be substituted in commands that take a group name as an argument.

handler. A software routine that controls a program's reaction to specific external events, such as an interrupt *handler*.

hardware. The equipment, as opposed to the programming, of a computer system.

header. Constant text that is formatted to be in the top margin of one or more pages.

header file. A text file that contains declarations used by a group of functions or users.

header label. A special set of records on a diskette describing the contents of the diskette.

help. Explanatory information that a program provides.

highlight. To emphasize an area on the display by any of several methods, such as brightening the area or reversing the color of characters within the area.

history file. A file containing a log of system actions and operator responses.

hole in a file. See *sparse file*.

home directory. (1) A directory associated with an individual user. (2) The user's current

directory on login or after issuing the **cd** command with no argument.

hook ID. A unique number assigned to a specific trace point. All trace entries include the hook ID of the originating trace point in the trace entry header. Pre-defined trace points use assigned hook IDs ranging from 0 to 299. User-defined trace points can choose hook IDs ranging from 300 to 399.

I/O. See *input/output.*

ID. Identification.

IF expressions. Expressions within a procedure, used to test for a condition.

informational message. A message providing information to the operator, that does not require a response.

initial program load (IPL). The process of loading the system programs and preparing the system to run jobs. See *initialize.*

initialize. To set counters, switches, addresses, or contents of storage to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

i-node. The internal structure for managing files in the system. I-nodes contain all of the information pertaining to the node, type, owner, and location of a file. A table of i-nodes is stored near the beginning of a file system.

i-number. A number specifying a particular i-node on a file system.

input. Data to be processed.

input device. Physical devices used to provide data to a computer.

input file. A file opened in the input mode.

input list. A list of variables to which values are assigned from input data.

input-output file. A file opened for input and output use.

input-output device number. A value assigned to a device by the virtual machine or to the virtual device by the IBM RT Virtual Resource Manager Technical Reference. This number uniquely identifies the device regardless of whether it is real or virtual.

input/output (I/O). Pertaining to either input, output, or both between a computer and a device.

input/output subsystem. That part of the VRM comprised of processes and device managers that provides the mechanisms for data transfer and I/O device management and control.

interactive processing. A processing method in which each system user action causes response from the program or the system. Contrast with *batch processing.*

interface. A shared boundary between two or more entities. An interface might be a hardware component to link two devices together or it might be a portion of storage or registers accessed by two or more computer programs.

interrupt. (1) To temporarily stop a process. (2) In data communications, to take an action at a receiving station that causes the sending station to end a transmission. (3) A signal sent by an I/O device to the processor when an error has occurred or when assistance is needed to complete I/O. An interrupt usually suspends execution of the currently executing program.

IPL. See *initial program load.*

job. (1) A unit of work to be done by a system. (2) One or more related procedures or programs grouped into a procedure.

job queue. A list, on disk, of jobs waiting to be processed by the system.

justify. To print a document with even right and left margins.

K-byte. See *kilobyte.*

kernel. The part of the AIX Operating System which participates in the control of computer functions such as input/output, management and control of hardware and software, and the scheduling of user processes.

key. A unique identifier (of type **key_t**) that names the particular interprocess communications member.

key pad. A physical grouping of keys on a keyboard (for example, numeric key pad, and cursor key pad).

keyboard. An input device consisting of various keys allowing the user to input data, control cursor and pointer locations, and to control the user-to-work station dialogue.

keylock feature. A security feature in which a lock and key can be used to restrict the use of the display station.

keyword. One of the predefined words of a programming language; a reserved word.

kill. An AIX Operating System command that stops a process.

kill character. The character that is used to delete a line of characters entered after the user's prompt.

kilobyte. 1024 bytes.

kprocs. A kernel parameter establishing the maximum number of processes that the kernel can run simultaneously.

label. (1) The name in the disk or diskette volume table of contents that identifies a file. See also *file name.* (2) The field of an instruction that assigns a symbolic name to the location at which the instruction begins, or such a symbolic name.

left margin. The area on a page between the left paper edge and the leftmost character position on the page.

left-adjust. The process of aligning lines of text at the left margin or at a tab setting such that the leftmost character in the line or filed is in the leftmost position. of the field is in the leftmost position. Contrast with *right-adjust.*

lexical analyzer. A program that analyzes input and breaks it into categories, such as numbers, letters or operators.

library. A collection of functions, calls, subroutines, or other data.

licensed program product (LPP). Software programs that remain the property of the manufacturer, for which customers pay a license fee.

linefeed. An ASCII character that causes an output device to move forward one line.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

load. (1) To move data or programs into storage. (2) To place a diskette into a diskette drive, or a magazine into a diskette magazine drive. (3) To insert paper into a printer.

loader. A program that reads run files into main storage, thus preparing them for execution.

local. Pertaining to a device directly connected to your system without the use of a communications line. Contrast with *remote.*

log. To record; for example, to log all messages on the system printer.

log in (v). To sign on at a work station.

log off (v). To sign off at a work station.

logical device. A file for conducting input or output with a physical device.

loop. A sequence of instructions performed repeatedly until an ending condition is reached.

main storage. The part of the processing unit where programs are run.

mapped file. A file that can be accessed using direct memory operations, rather than having to read it from disk each time it is accessed.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

master dump table. A structure containing dump table entries generated by VRM components. The dump program uses this table to locate data structures that should be included in a dump.

matrix. An array arranged in rows and columns.

maxprocs. A kernel parameter establishing the maximum number of processes that can be run simultaneously by a user.

memory. Storage on electronic chips. Examples of memory are random access memory, read only memory, or registers. See *storage.*

memory areas. Arrays of characters in memory.

menu. A displayed list of items from which an operator can make a selection.

message. (1) A response from the system to inform the operator of a condition which may affect further processing of a current program. (2) An error indication, or any brief information that a program writes to standard error or a queue. (3) Information sent from one user in a multiuser operating system to another. (4) A general method of communication between two processes.

message queue ID. An identifier assigned to a message queue for use within a particular process. It is similar in use to a *file descriptor* of a file.

message services. A set of routines to help create, update and display messages from a program.

minidisk. A logical division of a fixed disk that may be further subdivided into one or more partitions. See *partitions.*

modem. See *modulator-demodulator.*

modulation. Changing the frequency or size of one signal by using the frequency or size of another signal.

modulator-demodulator (modem). A device that converts data from the computer to a signal that can be transmitted on a communications line, and converts the signal received to data for the computer.

module. A discrete programming unit that usually performs a specific task or set of tasks. Modules are subroutines and calling programs that are assembled separately, then linked to make a complete program.

mount. To make a file system accessible by adding it to the logical file tree hierarchy.

mountab. A kernel parameter establishing the maximum number of file systems that can be mounted simultaneously.

msqid. See *message queue ID.*

multiprogramming. The processing of two or more programs at the same time.

multivolume file. A diskette file occupying more than one diskette.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (the nesting subroutine).

network. A collection of products connected by communication lines for information exchange between locations.

IBM AIX/RT Network File System (NFS). A licensed program that allows users to access and work with files located on other network computers without regard to machine type or operating system.

new-line character. A control character that causes the print or display position to move to the first position on the next line.

null. Having no value, containing nothing.

null character (NUL). The character hex 00, used to represent the absence of a printed or displayed character.

numeric. Pertaining to any of the digits 0 through 9.

object code. Machine-executable instruction, usually generated by a compiler from source code written in a higher level language. Consists of directly executable machine code. For programs that must be linked, object code consists of relocatable machine code.

octal. A base eight numbering system.

open. To make a file available to a program for processing.

operating system. Software that directs and controls the hardware and software in the computer system in which the operating system resides, by providing services such as resource allocation, scheduling, input/output control, hardware control, and data management.

operation. A specific action (such as move, add, multiply, load) that the computer performs when requested.

operator. A symbol representing an operation to be done.

output. The result of processing data.

output devices. Physical devices used by a computer to present data to a user.

output file. A file that is opened in either the output mode or the extend mode.

override. (1) A parameter or value that replaces a previous parameter or value. (2) To replace a parameter or value.

overwrite. To write output into a storage or file space that is already occupied by data.

owner. The user who has the highest level of access authority to a data object or action, as defined by the object or action.

pad. To fill unused positions in a field with dummy data, usually zeros or blanks.

page. A block of instructions, data, or both.

pagination. The process of adjusting text to fit within margins and/or page boundaries.

paging. The action of transferring instructions, data, or both between real storage and external page storage.

paging space. An area on disk that the system uses to store information that is resident in virtual memory, but is not currently being accessed.

pane. In **extended curses**, an area of the display that shows all or a part of the data contained in a presentation space associated with that pane. A pane is a subdivision of a panel.

panel. In **extended curses**, a rectangular area on the display consisting of one or more panes that a program can treat as a unit.

parallel processing. The condition in which multiple tasks are being performed simultaneously within the same activity.

parameter. Information that the user supplies to a panel, command, or function.

parent. (1) Pertaining to a secured resource, either a file or library, whose user list is shared with one or more other files or libraries. Contrast with *child*. (2) Also pertains to a

process that has **fork**ed to create one or more child processes.

parent directory. The directory one level above the current directory.

parser. A program that analyzes input and determines what to do with the input.

partition. A logical division of a fixed disk.

password. A string of characters that, when entered along with a user identification, allows an operator to sign on to the system.

password security. A program product option that helps prevent the unauthorized use of a display station, by checking the password entered by each operator at sign-on.

path name. A complete file name specifying all directories leading to that file.

pattern-matching character. Special characters such as * or ? that can be used in a search pattern. Some are used in a file specification to match one or more characters. For example, placing a ? in a file specification means any character can be in that position. Pattern-matching characters are also called *wildcards*.

permission code. A three-digit octal code indicating the access permissions. The access permissions are read, write, and execute.

permission field. One of the three-character fields within the permissions column of a directory listing indicating the read, write, and run permissions for the file or directory owner, group, and all others.

physical device. See *device*.

physical file. An indexed file containing data for which one or more alternative indexes have been created.

physical record. (1) A group of records recorded or processed as a unit. Same as *block*. (2) A unit of data moved into or out of the computer.

PID. See *process ID*.

pipe. To direct the data from one process to another process.

pipeline. A direct, one-way connection between two or more processes.

pitch. A unit of width of typewriter type, based on the number of times a letter can be set in a linear inch. For example, 10-pitch type has 10 characters per inch.

platen. The support mechanism for paper on a printer, commonly cylindrical, against which printing mechanisms strike to produce an impression.

port. (1) An access point for data entry or exit. (2) To run a program on more than one computer without modifying it for differences in operating systems.

portmap service (portmapper). An RPC daemon program that maps RPC services to port numbers to determine the servers that service the remote procedure requests made by clients.

position. The location of a character in a series, as in a record, a displayed message, or a computer printout.

presentation space. In **extended curses**, the data and attribute array associated with a window.

primary group. In concurrent groups, the group that is assigned to the files that you create.

print queue. A file containing a list of the names of files waiting to be printed.

printing device. Any printer or device that prints, such as a typewriter-like device or a plotter.

printout. Information from the computer produced by a printer.

priority. The relative ranking of items. For example, a job with high priority in the job queue will be run before one with medium or low priority.

privileged instructions. System control instructions that can run only in the processor's privileged state (VRM mode). Privileged instructions generally manipulate the virtual machines or the memory manager; they are typically not used by application programmers. See *privileged state.*

privileged state. A hardware protection state in which the processor can run privileged instructions.

problem determination. The process of identifying why the system is not working. Often this process identifies programs, equipment, data communications facilities, or user errors as the source of the problem.

problem determination procedure. A prescribed sequence of steps aimed at recovery from, or circumvention of, problem conditions.

procedure. See *shell procedure.*

process. (1) A sequence of actions required to produce a desired result. (2) An entity receiving a portion of the processor's time for executing a program. (3) An activity within the system begun by entering a command, running a shell program, or being started by another process.

process ID (PID). A unique number assigned to a process that is running.

profile. (1) A file containing customized settings for a system or user (2) Data describing the significant features of a user, program, or device.

program. A file containing a set of instructions conforming to a particular programming language syntax.

prompt. A displayed request for information or operator action.

propagation time. The time necessary for a signal to travel from one point on a communications line to another.

protocol. A set of rules for handling communications at the physical or logical level. Protocols often use other protocols to provide services.

queue. A line or list formed by items waiting to be processed.

queued message. A message from the system that is added to a list of messages stored in a file for viewing by the user at a later time. This is in contrast to a message that is sent directly to the screen for the user to see immediately.

quit. A key, command, or action that tells the system to return to a previous state or stop a process.

random access. An access mode in which records can be read from, written to, or removed from a file in any order.

real memory. Memory that is physically present in the system. Contrast with *virtual memory.*

recovery procedure. (1) An action performed by the operator when an error message appears on the display screen. Usually, this action permits the program to continue or permits the operator to run the next job. (2) The method of returning the system to the point where a major system error occurred and running the recent critical jobs again.

recursion. The process of using a function to define itself.

redirect. To divert data from a process to a file or device to which it would not normally go.

regular expression. A set of characters, metacharacters and operators that define a string or group of strings in a search pattern.

relational expression. A logical statement describing the relationship (such as greater than or equal) of two arithmetic expressions or data items.

relational operator. The reserved words or symbols used to express a relational condition or a relational expression.

relative address. An address specified relative to the address of a symbol. When a program is relocated, the addresses themselves will change, but the specification of relative addresses remains the same.

relative addressing. A means of addressing instructions and data areas by designating their locations relative to some symbol.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory.

remote. Pertaining to a system or device that is connected to your system through a communications line. Contrast with *local.*

Remote Procedure Call (RPC). An interface that allow remote communication with function call semantics. RPC provides a standard protocol for initiating and controlling processes on remote systems.

reserved word. A word that is defined in a programming language for a special purpose, and that must not appear as a user-declared identifier.

reset. To return a device or circuit to a clear state.

restore. Return to an original value or image. For example, to restore a library from diskette.

right adjust. The process of aligning lines of text at the right margin or tab setting such that the right-most character in the line or file is in the right-most position.

right-adjust. To place or move an entry in a field so that the rightmost character of the field is in the rightmost position. Contrast with *left-adjust.*

right margin. The area on a page between the last text character and the right upper edge.

root. Another name sometimes used for superuser.

root directory. The top level of a tree-structured directory system.

root file system. The basic AIX Operating System file system, which contains operating system files and onto which other file systems can be mounted. The root file system is the file system that contains the files that are run to start the system running.

routine. A set of statements in a program causing the system to perform an operation or a series of related operations.

RPC. See *Remote Procedure Call.*

RPCL. The input language to remote procedure calls made through RPC.

run. To cause a program, utility, or other machine function to be performed.

run-time environment. A collection of subroutines and shell variables that provide commonly used functions and information for system components.

SCCS. See *Source Code Control System.*

SCCS identification. In SCCS, a number assigned to a version of a program to keep track of each version of the program.

scratch file. A file, usually used as a work file, that exists until the program that uses it ends.

screen. (1) See *display screen.* (2) In **extended curses**, a special type of window that is as large as the terminal screen.

scroll. To move information vertically or horizontally to bring into view information that is outside the display or pane boundaries. to bring into view information that is outside the display's boundaries.

second level interrupt handler (SLIH). A routine that handles the processing of an interrupt from a specific adapter. An SLIH is called by the first level interrupt handler associated with that interrupt level.

sector. (1) An area on a disk track or a diskette track reserved to record information. (2) The smallest amount of information that can be written to or read from a disk or diskette during a single read or write operation.

security. The protection of data, system operations, and devices from accidental or intentional ruin, damage, or exposure.

segment. A contiguous area of virtual storage allocated to a job or system task. A program segment can be run by itself, even if the whole program is not in main storage.

segment registers. Registers in the system that hold the actual addresses of the memory segments currently in use.

semaphore. A general method of communication between two processes that is an extension of the features of signals.

semaphore ID. An integer that points to a set of semaphores and a data structure that contains information about the semaphores.

semid. See semaphore ID

serialize. To encode or convert the internal representation of data to a uniform or standard format. See *XDR.*

separator. A character used to separate parts of a command. See *delimiter.*

server. A computer that contains the data, services, or resources that can be accessed by other computers or processes on the network.

sequential access. An access method in which records are read from, written to, or removed from a file based on the logical order of the records in the file.

shared memory. An area of memory that more than one cooperating process can access simultaneously.

shared memory ID. An identifier assigned to the shared segment for use within a particular process. It is similar in use to a *file descriptor* of a file.

shared printer. A printer that is used by more than one work station.

shell. See *shell program.*

shell procedure. A series of commands combined in a file that carry out a particular function when the file is run or when the file is specified as an argument to the **sh** command. Shell procedures are frequently called shell scripts.

shell program. A program that accepts and interprets commands for the operating system (there is an AIX shell program and a DOS shell program).

shmid. See shared memory ID.

sign off. To end a session at a display station.

sign on. To begin a session at a display station.

sign-off. The action an operator uses at a display station to end working at the display station.

sign-on. The action an operator uses at a display station to begin working at the display station.

signal. A simple method of communication between two processes.

socket. A port identifier that associates an Internet address with a port.

software. Programs.

sort. To select a particular group of records from a file based upon some criterion. Also, to rearrange some or all of a group of records based upon items in a particular field of those records.

Source Code Control System (SCCS). A program for maintaining version control for the source files of a developing program.

source diskette. The diskette containing data to be copied, compared, restored, or backed up.

source program. A set of instructions written in a programming language, that must be translated to machine language compiled before the program can be run.

sparse file. A file that is created with a length greater than the data it contains, leaving empty spaces for future addition of data.

special character. A character other than an alphabetic or numeric character. For example; *, +, and % are special characters.

special file. Special files are used in the AIX system to provide an interface to input/output devices. There is at least one special file for each device connected to the computer. Contrast with directory and ordinary AIX files.

stand-alone work station. A work station that can be used to perform tasks independent of (without being connected to) other resources such as servers or host systems.

standard error. The place where many programs place error messages.

standard input. The primary source of data going into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can be to a file or another command.

standard screen. In **extended curses**, a memory image of the screen that the routines make changes to.

stanza. A group of lines in a file that together have a common function. Stanzas are usually separated by blank lines, and each stanza has a name.

statement. An instruction in a program or procedure.

status. (1) The current condition or state of a program or device. For example, the status of a printer. (2) The condition of the hardware or software, usually represented in a status code.

stderr. See *standard error.*

stdin. See *standard input.*

stdout. See *standard output.*

storage. (1) The location of saved information. (2) In contrast to memory, the saving of information on physical devices such as disk or tape. See *memory.*

storage device. A device for storing and/or retrieving data.

stream. Sequential input or output from an open file descriptor.

string. A linear sequence of entities such as characters or physical elements. Examples of strings are alphabetic string, binary element string, bit string, character string, search string, and symbol string.

subdirectory. A directory contained within another directory in the file system hierarchy.

subprogram. A program invoked by another program. Contrast with *main program.*

subroutine. (1) A sequenced set of statements that may be used in one or more computer programs and at one or more points in a computer program. (2) A routine that can be part of another routine.

subscript. An integer or variable whose value refers to a particular element in a table or an array.

substring. A part of a character string.

subsystem. A secondary or subordinate system, usually capable of operating independently of, or synchronously with, a controlling system.

superblock. The most critical part of the file system containing information about every allocation or deallocation of a block in the file system.

superuser. The system user with superuser privileges.

superuser priviledges. The unrestricted ability to access and modify any part of the operating system associated with the user who manages the system.

system. The computer and its associated devices and programs.

system call. A request by an active process for a service by the system kernel.

system customization. A process of specifying the devices, programs, and users for a particular data processing system.

system date. The date assigned by the system user during setup and maintained by the system.

system dump. A printout of storage from all active programs (and their associated data) whenever an error stops the system. Contrast with *task dump.*

system profile. A file containing the default values used in system operations.

system unit. The part of the system that contains the processing unit, the disk drives, and the diskette drives.

system user. A person who uses a computer system.

target diskette. The diskette to be used to receive data from a source diskette.

task. A basic unit of work to be performed. Examples are a user task, a server task, and a processor task.

task dump. A printout of storage from a program that failed (and its associated data). Contrast with *system dump.*

TCP/IP. See *Transmission Control Protocol/Internet Protocol.*

terminal. (1) An input/output device containing a keyboard and either a display device or a printer. Terminals usually are connected to a computer and allow a person to interact with a computer. See *work station.* (2) In **extended curses**, a memory image of what the terminal screen currently looks like.

text. A type of data consisting of a set of linguistic characters (for example, alphabet, numbers, and symbols) and formatting controls.

text application. A program defined for the purpose of processing text data (for example, memos, reports, and letters).

token. (1) The smallest independent unit of meaning as defined by either the parser or the

lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of a language syntax. (2) In M4, any string of letters and digits that **m4** recognizes.

token numbers. Nonnegative integers that represent the names of tokens.

trace. To record data that provides a history of events occurring in the system.

trace entry. A data structure containing a header of identifying information plus up to 20 bytes of defined data. Trace entries are generated by trace points and written to a trace log file.

trace point. A group of code statements that generates a trace entry from within a software program. Trace points are assigned to an event class which can be active or inactive. Trace points with active event classes can generate trace entries.

trace profile. An ASCII file that can be modified to activate or deactivate the various event classes. The trace profile is used by the trace daemon to set up three channel tables that show which event classes are active.

trace template. Used by the trace formatter to determine how the data contained in a trace entry should be formatted. All trace templates are stored in the master template file.

track. A circular path on the surface of a fixed disk or diskette on which information is magnetically recorded and from which recorded information is read.

Transmission Control Protocol/Internet Protocol (TCP/IP). The data transport protocol that provides for asynchronous data flow (streams) between hosts. TCP/IP allows a message to span more than one packet while ensuring the data is delivered completely, correctly, and in sequence. Contrast with *User Datagram Protocol/Internet Protocol.*

trap. An unprogrammed, hardware-initiated jump to a specific address. Occurs as a result of an error or certain other conditions.

tree-structured directories. A method for connecting directories such that each directory is listed in another directory except for the root directory, which is at the top of the tree.

truncate. To shorten a field or statement to a specified length.

typematic key. A key that repeats its function multiple times when held down.

type style. Characters of a given size, style, and design.

Uid. See *user number.*

User Datagram Protocol/Internet Protocol (UDP/IP). A data transport protocol that sends individual packets of data between hosts. UDP/IP is a simple protocol that does not guarantee reliable results.

user ID. See *user number.*

user name. A name that uniquely identifies a user to the system.

user number (Uid). A unique number identifying an operator to the system. This string of characters limits the functions and information the operator is allowed to use. The Uid can often be substituted in commands that take a user's name as an argument.

user profile. A file containing a description of user characteristics and defaults (for example, printer assignment, formats, group ID) to be conveyed to the system while the user is signed on.

utility. A service; in programming, a program that performs a common service function.

valid. (1) Allowed. (2) True, in conforming to an appropriate standard or authority.

value. (1) In Usability Services, information selected or typed into a pop-up. (2) A set of

characters or a quantity associated with a parameter or name. (3) In programming, the contents of a storage location.

variable. A name used to represent a data item whose value can change while the program is running. Contrast with *constant.*

verify. To confirm the correctness of something.

version. Information in addition to an object's name that identifies different modification levels of the same logical object.

virtual device. A device that appears to the user as a separate entity but is actually a shared portion of a real device. For example, several virtual terminals may exist simultaneously, but only one is active at any given time.

virtual machine. The hardware dependent portion (kernel, shells, libraries, and other subsystems) of the AIX Operating System and user applications.

virtual machine interface (VMI). A standard software interface between the kernel and the VRM.

virtual memory. Addressable space that appears to be real memory. From virtual memory, instructions and data are mapped into real memory locations. Contrast with *real memory.*

Virtual Resource Manager (VRM). A portion of the AIX Operating System that provides various services, interfaces, and runtime routines, through which AIX controls the IBM RT hardware and peripherals.

virtual storage. See virtual memory

Volume ID (Vol ID). A series of characters recorded on the diskette used to identify the diskette to the user and to the system.

VRM. See *virtual resource manager.*

wildcard. See *pattern matching characters.*

window. In **extended curses**, a memory image of what a section of the terminal screen looks like at some point in time. A window can be either the entire terminal screen, or any smaller portion down to a single character.

word. A contiguous series of 32 bits (four bytes) in storage, addressable as a unit. The address of the first byte of a word is evenly divisible by four.

work file. A file used for temporary storage of data being processed.

work station. A device at which an individual may transmit information to, or receive information from, a computer for the purpose of performing a task, for example, a display station or printer.

working directory. See *current directory.*

eXternal Data Representation (XDR). A data definition language that is used to represent internal machine data types in a uniform or standard format across a network of machines. The representation of objects in the machine data are standardized to facilitate remote communications among machines.

Index

Special Characters

.DEFAULT 2-35
.SUFFIXES 2-30
$< macro 2-38
$$@ macro 2-38
$* macro 2-38
$% macro 2-38
$? macro 2-38
$@ macro 2-38
2-25
_doprnt subroutine E-4

A

about this book iii
absolute value 3-33
adb
adb, program debugger 8-64
 core image 8-67
 data files 8-67
 displaying instructions 8-69
 input format 8-108
 locating values 8-113
 memory maps 8-100
 patching binary files 8-113
 prompt option 8-68
 starting 8-64, 8-67
 stopping 8-64
 write option 8-68
 writing to a file 8-113
admin command, using 10-9
apply list file 9-17, 9-34
archive control file 9-35
archive libraries, with make program 2-33
ARGSUSED 2-11
arguments, SCCS command 10-6
arrays 11-15
as command 2-20
 using 2-20
assembler language 2-20
assembler listing 2-20
assembling source file 2-20
audience of this book iii
authentication
 RPC message authentication 15-4
awk 1-4
 action 11-7
 arrays 11-15
 control statements
 break 11-17
 continue 11-17
 exit 11-17
 for 11-17
 if-else 11-16
 next 11-17
 while 11-16
 defined variables 11-8
 field separator 11-8
 field variables 11-14
 functions 11-12
 macros 11-8
 operation 11-7
 operators 11-14
 program file 11-6
 syntax 11-7
 record separator 11-8
 regular expressions 11-9
 relational expressions 11-11
 search pattern 11-6
 defining 11-7
 special characters 11-10
 strings
 concatenating 11-15
 variables 11-13

B

backslash (\) 2-25
Berkeley subroutines E-1
 differences in include files in AIX E-7
 differences in routines in AIX E-4
 include files E-6
 ioctls E-8
 list of library subroutines E-2
 porting applications to AIX E-5
 subroutines E-1
 tty devices E-7
Berkeley 4.3 E-1
 differences in include files in AIX E-7
 differences in routines in AIX E-4
 include files E-6
 ioctls E-8
 porting applications to AIX E-5
 subroutines E-1
 tty devices E-7
bessel functions 3-33
bffcreate 9-7
binary tree 3-26
bit fields 2-13
BNU 14-16, 14-17
body, in SCCS file 10-6
branch, SID 10-5
break statement 2-7
breakpoints 8-92
broadcast RPC 15-37
building programs 2-22
 description file 2-22
 macros 2-22
 operation 2-22
 parent file 2-23
 rules 2-23
 target file 2-22

C

C
 library functions 3-8
 operator precedence 2-15
 program checking 2-5
 data type 2-7
 external names 2-13
 function definitions 2-9
 functions 2-10
 initializing variables 2-12
 portability 2-13
 structure 2-9
 union 2-9
 use of characters 2-13
 variables 2-10, 2-12
C language
 libraries
 c library 1-10
 extended curses 1-10
 math library 1-10
 run time services library 1-10
 stdio 1-10
 macro preprocessor, m4 12-2
C language book iv
calls
 error reporting 4-5
 file maintenance 4-83
 file system 4-78
 header files 4-2
 include files 4-2
 memory management 4-77
 message 4-55
 operation 4-57
 sample program 4-62
 pipe 4-13
 process
 exec 4-4
 exit 4-4
 fork 4-4
 nice 4-4
 pipe 4-4
 plock 4-4
 process ID 4-18
 sample program 4-23

process tracking 4-29
reference book iv
semaphore 4-42
operation 4-44
sample program 4-49
structures 4-42
shared memory 4-74
signal
sample program 4-31
signals 4-29
time 4-84
wait 4-10
casts 2-10
catclose 16-12
catgetmsg 16-13
catgets 16-13
catopen 16-11
cc command 2-3, 2-4
examples 2-4
using
for assembler language 2-21
for c programs 2-4
what it does 2-4
changing a program 9-2
changing strings 11-7
channels tables, trace 7-10
character
header file 3-23
library functions 3-18
character collation 14-4
character conversion 14-4, 14-5
character range 14-4
characters
use in a C program 2-13
checksum, in SCCS header 10-6
child process 4-5
chkcomp 9-7
chngstate 9-7
ckprereq 9-8
client
code page 14-3
code point 14-3, 14-4
collating sequence 14-4
definition 14-4
equivalence class 14-4
extended character 14-3
codes, printer
See printer codes
collation
character 3-22
equivalence class 3-19, 3-23
extended 3-22
NCcollate 3-22
NCcoluniq 3-22
NCeqvmap 3-23
NLgetctab 3-14
command
cc 2-3, 2-4
installation 9-6
installation, internal 9-7
installp 9-6
command conventions, SCCS 10-6
commands 14-5, 14-16, 14-17, 14-18
adduser 14-16
as 2-20
BNU 14-17
ctab 14-5
dump, summary of 7-45
error log, summary of 7-27
ld 1-8, 2-21
reference book iv
SCCS, summary of 10-7
shlib 1-5, 1-8
trace, summary of 7-8
uucico 14-18
uuclean 14-18
uucp 14-18
uulog 14-18
uuname 14-18
uupick 14-18
uusub 14-18
uuto 14-18
uux 14-18
uuxqt 14-18
comments 2-25
comments 2-25
comments, in SCCS header 10-6
compiler 2-3
compiling 2-2
component dump table 7-42
compressed printing C-6
config 9-17

configuration
error log file 7-26
trace log file 7-7
trace profile 7-7
converting numbers 3-24
converting 4.3BSD application programs to AIX E-5
creating files 2-23
creating shared library 1-5
ctab command 14-5
currency 14-5
format 14-5
symbol 14-5
curses 5-2
cursescontrolling display screen
See extended
cursesdisplay screen
See extended
cursesscreen handling
See extended
curseswriting to display screen
See extended
cur00.h 5-5
cur01.h 5-5
cur05.h 5-5
cvid 9-8

D

data type 2-7
array 2-8
array pointer 2-8
casts 2-10
checking
turning off 2-9
mixing 2-8
date input 14-5
dbx
child processes 8-28
command files 8-35
controlling program execution
breakpoints, setting and deleting 8-14
overview 8-14
core files 8-11, 8-26
current file 8-26, 8-27
changing 8-27
displaying 8-26
debugging environment
.dbxinit file 8-34
alias facility 8-33
changing the prompt 8-34
custom commands 8-33
custom initialization file 8-34
dbx primitives 8-33
debugging programs in progress 8-12
default editor 8-28
event IDs 8-19
expressions
declarations 8-23
disabling type checking 8-24
displaying 8-21
modifying 8-21
overriding type checking 8-23
printing 8-21
type 8-23
type checking 8-23
type renaming 8-23
valid operators 8-23
help facility 8-36
machine debugging
displaying registers 8-30
examining memory addresses 8-31
memory addresses 8-31
modes 8-31
overview 8-30
printing memory 8-31
registers 8-30
running at machine level 8-32
tracing execution 8-32
multiple process debugging 8-28
overview 8-5
printing
changing print output 8-25
procedures
calling 8-20
changing the current 8-27
listing 8-20
passing arguments 8-20
stack tracing 8-20
program forks 8-28

running your program
continuing execution 8-16
input 8-17
one line at a time 8-16
output 8-17
redirecting input 8-16
system calls 8-17
the run command 8-16
scoping names 8-22
sdb equivalencies 8-36
searching for a string 8-27
shell commands 8-35
signal handling 8-19
signals 8-19
source directory path
changing 8-26
source files 8-26
special variables 8-25
starting dbx 8-11, 8-12
samples of 8-12
stop events 8-19
summary of commands 8-6-8-11
symbolic debugging 8-13
text editors 8-28
text strings 8-27
the debug flag 8-14
tracing execution 8-18, 8-19
variables
displaying 8-21
folding 8-24
modifying 8-21
printing 8-21
xdbx
.Xdefaults file 8-41, 8-42
automatic windows 8-37
changing the environment 8-42
command window 8-37
custom commands 8-44
dual 8-40
environment 8-42
format menu window 8-38
instruction menu window 8-38
instruction window 8-38
machine 8-40
message window 8-39
modes 8-40
register window 8-39
scroll bar 8-39
selecting attributes 8-40
selecting menu items 8-41
source menu window 8-39
source mode 8-40
source window 8-39
specifying a mode 8-40
status window 8-40
using a mouse with xdbx 8-41
watch window 8-40
window environment 8-37
debug
See also dbx
dbx, multiple process 8-28
sdb 1-4
Debugger
See also adb
See also dbx
dbx, multiple process 8-28
Debugging
See also dbx
dbx, multiple process 8-28
decimal character 14-5
declarations
including from program 3-7
delivery protocol
delta command 10-4
delta command, using 10-16
delta table, in SCCS header 10-6
description file
colon, double 2-26
colon, single 2-25
command sequences 2-25
commands 2-25
contents 2-24
example 2-28
format 2-24
line continuation 2-25
macro 2-37
simplifying 2-29
directories 3-7, 3-14, 3-23, 3-24, 3-26, 3-30, 3-32
/usr/include 3-7, 3-14, 3-23, 3-24, 3-32
/usr/lib 3-30
current 3-26
disk space saving 1-5

Distributed Services
IPC queues 3-34
DOS Services
books iv
double strike printing C-6
double wide printing C-6
dspcat 16-20
dspmsg 16-20
dump
commands, using 7-45
component dump table 7-44
dump diskette 7-44
dump formatter 7-44
initiation by VRM 7-44
master dump table 7-44
report, example 7-45
restore command 7-44
dump components, diagram of 7-42
dump data 7-42
dump diskette 7-42
dump facilities 7-42
dump table entry 7-42

E

emphasized printing C-6
enumerated data type 2-9
enumerator 2-9
environment variables 2-44, 14-3, 14-10, 14-11, 14-18
LANG 14-11
NLCTAB 14-10
NLCURSYM 14-10
NLDATE 14-10, 14-18
NLFILE 14-10
NLLANG (AIX) 14-10
NLLDATE 14-10
NLLDAY 14-10
NLLMONTH 14-10
NLSDAY 14-10
NLSMONTH 14-10
NLTIME 14-10
NLTMISC 14-10
NLTSTRS 14-10
NLTUNITS 14-11
NOSTR 14-11
used by make command 2-44
YESSTR 14-11
environment, system
extended characters 5-11
extended curses 5-10
errno 4-5
error entry 7-23
error ID 7-23
error identifier 7-23
error log
class 7-29
commands, using 7-27
data_descriptor 7-37
definition 7-37
device driver 7-25
error daemon 7-25
error ID 7-29
error identifier 7-29
error log file 7-25
errsave, example 7-34
format file 7-25
log file, altering the 7-26
mask 7-29
match values, using 7-38
output data, formatting 7-39
report, example 7-40
subclass 7-29
subroutines, using 7-29, 7-31
template, example 7-40
templates, creating 7-35
templates, defining 7-27
templates, syntax 7-35
templates, updating 7-28
type 7-30
error log components, diagram of 7-23
error log facilities 7-23
error messages
SCCS, format of 10-7
error point 7-23
errors, logging 7-1
errupdate 9-8
event class 7-4
exec 4-6, 4-12
sample program 4-12

executing a program 8-89
exit 4-6
exponential 3-33
extended character 14-6
extended curses 5-2, 5-5
 attributes 5-13
 boxes 5-13
 compiling a program 5-5
 controlling input 5-33
 curscr 5-4
 display attributes 5-27
 changing attributes 5-29
 environment 5-10
 setting up 5-12
 example program 5-36
 extended characters 5-4
 extended routine 5-33
 features 5-2
 field 5-4
 function names 5-9
 combining 5-9
 getting input 5-16
 header files 5-5
 initializing the screen 5-12
 insert functions 5-15
 keypad routine 5-33
 pane 5-4
 PANE structure B-6
 panel 5-4
 PANEL structure B-5
 panels 5-21
 panes 5-21
 linkage 5-22
 partial character indicator 5-4
 prerequisites 5-5
 presentation space 5-4
 programming structures B-1
 routine categories 5-3
 routines 5-3
 using 5-12
 screen 5-4
 screen appearance 5-7
 screen dimensions 5-7
 screen update 5-6
 stdscr 5-4
 system environment 5-10
 terminal 5-4
 terms 5-4
 trackloc routine 5-33
 variables 5-10
 what you need 5-5
 window 5-4
 WINDOW structure B-1
 windows 5-17
 scrolling 5-35
 writing to a window 5-13
External Data Representation (XDR)
 converting data types to a uniform representation 15-12
 deserializing data 15-13
 serializing data 15-12
 using to allocate memory 15-16
 XDR 15-12
external names 2-13

F

FF
 See form feed
file control 10-1
file system calls 4-78
file, SCCS 10-4
files 14-10, 14-16, 14-18
 /etc/passwd 14-16
 /usr/lib/nls/*.in 14-10
 accessing
 with library functions 3-11
 apply list 9-34
 archive control 9-35
 BNU log files 14-18
 branch delta, creating 10-14
 corrupted 10-10
 creating 10-9
 duplicate version, getting 10-15
 editable version, getting 10-13
 format, SCCS 10-5
 maintenance 4-83
 program history 9-26
 program name 9-33
 program requirements 9-29

read-only version, getting
recovering 10-10
release number, changing 10-13
SCCS, naming conventions 10-9
special requirement 9-36
status
with library functions 3-12
system calls 4-83
updating 10-16
using with system call 4-79
files, library description 2-15
finding strings 11-3
flags, SCCS command 10-6
fork 4-4
sample program 4-8
form feed C-4
format 14-5
date 14-5
input 14-5
output 14-5
time 14-5
Forms control, printer C-4
FORTRAN 14-7
function 2-10, 2-14
calling 2-7
calling another function 2-14
functions 2-9

G

gencat 16-10
get command, using 10-11
Graphics codes, printer C-7
grep 1-4
defining string patterns 11-4
extended grep (egrep) 11-3
fast grep (fgrep) 11-3
wildcard 11-4
group ID
effective 4-18
real 4-18

H

hash tables 3-26
header files 3-7, 3-8, 3-14, 3-15, 3-18, 3-23, 3-24, 3-25, 3-28, 3-32, 3-34, 4-2, 5-5, 14-7, 14-22
ctype.h 3-23
definition of 3-7
drs.h 3-34
extended curses 5-5
grp.h 3-25
math.h 3-32
memory.h 3-18
NLchar.h 14-22
NLctype.h 14-22
NLregexp.h 14-22
pwd.h 3-25
regexp.h 3-15, 14-7
signal.h 3-28
stdio.h 3-8, 3-14
time.h 3-23, 3-24
use of 3-7
help 6-1, 6-22
definition 6-2, 6-22
displaying 6-29
file path name
changing 6-23
changing for debug 6-24
default 6-23
format 6-22
header files 6-29
help file 6-25
building 6-25
contents 6-25
routines 6-22
using 6-22, 6-29
hook ID 7-4
hyperbolic functions 3-33

I

I/O
 display screen 1-10
 header file 3-14
 library functions 3-12, 3-13
ICANON 14-9
ID keywords 10-11
identification keywords 10-9, 10-11
 warning, getting files 10-17
include
 m4 built-in function 12-13
index, message 6-11
information in this book iii
init 4-7
initializing variables, C 2-12
input format 14-5
installation
 apply list file 9-34
 archive control file 9-35
 procedure 9-11, 9-18
 customization 9-12
 customizing 9-12
 quiescing the system 9-12
 restoring the program 9-11
 returning to installp 9-13, 9-18
 what it must do 9-11
 program history file 9-26
 program name file 9-33
 program requirements file 9-29
 special requirement file 9-36
 what you need 9-9
installation services 9-2
 using 9-6
installc 9-8
installing a program 9-2
installing Programming Examples A-1
installp 9-6, 9-13, 9-18
 return codes to 9-13
international character 14-16
 synonym 14-16
international character support 14-1, 14-3, 14-6, 14-7, 14-10, 14-18
 commands 14-18
 environment variables 14-10
 features 14-6
 initialization 14-6
 NLgetenv 14-6
 regular expression 14-7
 RTPC_NO_NLS 14-7
inudocm 9-8
inurecv 9-8
inurest 9-8
inusave 9-8
inuupdt 9-8
IPC queues 3-34

J

Japanese Language Support Operating System 3-2, 14-3

K

kernel 14-8

L

LANG environment variable 16-2
languages 2-3
 assembler 2-20
ld command 2-21
level, SID 10-5
lex 2-7
libraries 1-10, 3-1, 3-3
 accessing files 3-11
 bessel functions 3-33
 binary tree functions 3-26
 C library 3-8
 character functions 3-18
 header file 3-23
 converting numbers 3-24
 get status information 3-12
 getting system parameters 3-26

group access functions 3-25
hash table function 3-26
hyperbolic functions 3-33
including on command line 3-7
input functions 3-12
math 3-32
including on the command line 3-32
memory allocation functions 3-27
memory functions 3-18
network functions 3-34
output functions 3-13
password functions 3-25
pseudo-random number functions 3-27
random number functions 3-27
run-time services 3-30
signal functions 3-28
string functions 3-15
system 3-3
table management functions 3-27
time function header file 3-24
trigonometry functions 3-32
libraries, with make program 2-33
library control 10-1
library description files 2-15
library files 3-4, 3-5, 3-6, 3-15, 3-30
/lib/libc.a 3-4
/lib/libdossio.a 3-5
/lib/libfc.a 3-4
/lib/libfm.a 3-4
/lib/libm.a 3-4
/lib/libPW.a 3-4, 3-15
/lib/librts.a 3-4
/usr/lib/libcur.a 3-5
/usr/lib/libcurses.a 3-5
/usr/lib/libdbm.a 3-6
/usr/lib/libdos.a 3-5
/usr/lib/libg.a 3-6
/usr/lib/libipc.a 3-4
/usr/lib/libl.a 3-6
/usr/lib/libplot.a 3-6
/usr/lib/libprint.a 3-6
/usr/lib/libqb.a 3-6
/usr/lib/librem.a 3-6
/usr/lib/librts.a 3-30
/usr/lib/liby.a 3-6
/usr/lib/lib2.a 3-4
/usr/lib/lib300.a 3-5
/usr/lib/lib300s.a 3-5
/usr/lib/lib4014.a 3-5
/usr/lib/lib450.a 3-5
linking 2-2, 2-21
lint 2-5
command syntax 2-5
creating a library 2-15
flags 2-5
library description files 2-15
operation 2-6
starting 2-5
logarithm 3-33
login synonym 14-16

M

macro
defining 2-36
make's internal 2-38
precedence for make program 2-38
preprocessor, m4 12-2
make 1-4
.DEFAULT 2-35
adding suffixes 2-30
building programs 2-22
command syntax 2-23
description file 2-24
example 2-45
macros in a 2-37
description file example 2-28
environment variables 2-44
error handling 2-27
flags 2-23
functions 2-22
ignore errors 2-27
including other files 2-35
internal rules 2-29
default 2-35
libraries 2-33
macro 2-34
defining 2-36
internal 2-38
precedence 2-38

macro definitions 2-23
makefile 2-24
MAKEFLAGS 2-26
nested calls 2-26
parent file 2-24
prevent writing 2-27
rules file 2-34
rules file example 2-30
rules, internal
writing 2-34
rules, single suffix 2-33
shell commands 2-26
target file 2-23
using with SCCS files 2-43
write only flag 2-26
master dump table 7-42
math library functions 3-32
memory 3-27
allocation 3-27
library functions 3-18
memory management calls 4-77
memory saving 1-5
message calls 4-55
sample program 4-62
message facility
creating catalogs 16-10
displaying messages from a shell script 16-20
message text source files 16-3
comments 16-3
delset command 16-3
message text 16-3
quote command 16-5
set command 16-3
opening and closing catalogs 16-11
reading messages 16-12
using catalogs 16-10
message services
See messages
message table 6-8
messages
adding 6-10
definition 6-2
displaying 6-14
error number 6-5
example message table 6-8
format
immediate 6-4
queued 6-4
standard file 6-8
header files 6-13
immediate 6-3
format 6-4
generating 6-14
index 6-11
message table 6-8
program identifiers 6-6
queued 6-3
format 6-4
generating 6-14
sample message 6-5
severity code 6-5
symbols for variables 6-15
table
naming 6-10
text insert
example 6-20
text insert definition 6-11
time stamp 6-5
types of 6-3
using 6-13
variable fields in 6-15
example 6-16, 6-18, 6-20
metacharacter interpretation 14-4
mkcatdefs 16-7
moving a program 2-13
msghelp 6-29
msgrtrv 6-30
multiple processes
shared text 1-5
m4
function
divert 12-13
sinclude 12-13
undivert 12-13
functions
eval 12-12
include 12-13
incr 12-12
maketemp 12-14
syscmd 12-14
integer arithmetic 12-12

system command 12-14
m4 macro preprocessor
command syntax 12-3
defining macros 12-4
functions
arguments for 12-6
changequote 12-10
define 12-4
ifdef 12-11
undefine 12-11

N

natural logarithm 3-33
network
library functions 3-34
NL_MAXOPEN 16-2
NL_MSGMAX 16-2
NL_SETMAX 16-2
NL_TEXTMAX 16-2
NLcatgets 16-16
NLcatopen 16-12
NLchar 14-6
NLfile 14-6
NLgetamsg 16-17
NLgetenv subroutine 14-6
NLSPATH environment variable 16-2
numbers
converting to other forms 3-24

O

Operating system
reference books for iv
using, books for iv
writing device drivers iv
options, in SCCS header 10-6
output format 14-5

P

panels B-5
panes B-6
parent process 4-4
password
library functions 3-25
password file 3-25
pios
See printer
pipe 4-13
sample program 4-13
portability 2-13
bit fields 2-13
external names 2-13
porting application programs
from 4.3BSD to AIX E-5
power, raising to a 3-33
precedence, operator 2-15
printer
codes, control C-1
data stream C-1
printer control codes
graphics C-7
page appearance C-4
paper control C-4
print mode C-6
printhead C-3
ribbon control C-5
type style C-6
printing C-2
codes, control
ASCII C-2
hexadecimal C-2
Keys to generate C-2
Names C-2
I/O support C-2
miscellaneous C-3
printing ASCII codes less than 32 C-3
process 4-4
group 4-18
ID 4-18
special IDs 4-7
process group 4-18
process ID 4-18

program
changes 9-2
installation 9-2
updating 9-2
program checking, C 2-5
program control 10-1
program file 8-66
program history file 9-26
program name file 9-33
program requirements file 9-29
example entry 9-31
programming languages 2-3
programming tools
cb 1-3
cc 1-4
cflow 1-3
cxref 1-3
ed 1-3
lint 1-3
programming with remote procedure calls 15-3
library routines 15-18
programming levels 15-18
programs, monitoring 7-1
pseudo-random number generator 3-27

R

random number generator 3-27
redirecting I/O 4-9
regular expressions
definition 11-4, 11-21
used with awk 11-9
related books iv
related information iv
relational expressions 11-11
release, SID 10-5
Remote Procedure Call (RPC)
high level
intermediate level
low level
Remote Procedure Call Language (RPCL) 15-6
defining data types 15-6
input language 15-6
remote procedure calls 15-3
authentication 15-4
broadcast 15-37
callrpc routine 15-20
communication model 15-3
input language 15-6
registerrpc routine 15-21
terms 15-3
RPC
See Remote Procedure Call (RPC)
RPC protocol 15-4
assigning procedure numbers 15-5
assigning program numbers 15-5
assigning version numbers 15-5
RT
programming tools 1-2
system services 1-2
RTPC_NO_NLS variable 14-7
rules
changing, make 2-34
rules, make program 2-29, 2-30
rules, single suffix 2-33
runcat 16-9

S

sample programs
c source files 1-11
compiling 1-11
printing 1-11
scanning files 11-6
sccs 1-4, 2-33
admin, using 10-9
command conventions 10-6
commands
summary of 10-7
delta, using 10-16
error messages, format of 10-7
features 10-3
file format 10-5
files
branch delta, creating 10-14
duplicate version, getting 10-15
editable version, getting 10-13
naming conventions 10-9

read-only version, getting
recovering 10-10
release number, changing 10-13
warning, non-SCCS commands 10-6
get, using 10-11
identification keywords 10-9
overview 10-3
terminology 10-4
using make with 2-43
description files 2-44
SCCS identification 10-4
scrolling 5-35
sdb 1-4
search algorithm 3-26
searching
tables 3-27
sed
defining string patterns 11-21
wildcard 11-21
see awk.changing strings 11-3
select procedure 15-35
semaphore 4-42
sample program 4-49
structures 4-42
sequence, SID 10-5
server
setrdperm 9-7
shared libraries 1-5
shared library
changing 1-8
installing 1-6
keys 1-6
linking 1-8
shared memory calls 4-74
sharp 2-25
shell commands
in a c language program 1-9
shlib 1-5
SID 10-4
signal 4-30, 8-92
handling 4-30
sample program 4-31
trapping 4-31
signals
library functions 3-28
software generated 3-28
single-shift byte 14-3
SNA 14-16
Source Code Control System 10-1
special requirement file 9-36
change configuration 9-37
change report templates 9-37
change VRM 9-38
dependent program 9-36
ipl required 9-37
square root 3-33
stdio 3-14
storage
getting 3-27
string 3-9, 3-10, 3-11, 3-12, 3-13, 3-15, 3-16, 3-17, 3-19, 3-23, 3-24, 3-29
ASCII 3-15, 3-19
char to NLchar 3-15
character search 3-15
compare 3-16
compiled regular expression 3-15
concatenate 3-15, 3-16
conversion 3-24, 3-29
convert 3-17
converting to other forms 3-24
copy 3-16, 3-17
data type 3-9
date and time 3-23, 3-24
definition 11-3
format from 3-12
format to 3-13
from stdin 3-12
from stream 3-12
functions 3-10, 3-11
dnl (delete to new-line) 12-17
dumpdef 12-17
errprint 12-17
index 12-16
substr 12-16
translit 12-16
header file 3-15, 3-23
length 3-16, 3-17
library 3-15
library functions 3-15
literal 11-4
manipulation functions 3-15
NLchar to char 3-15

printing 12-17
regular expression 3-15
regular expression to 3-15
search 3-17
single quotes with 11-3
strings 12-16
to stdout 3-13
to stream 3-13
using format 3-13
with m4 12-16
strings
changing 11-3
concatenating, in awk 11-15
finding 11-3
structures 2-9
subroutines 14-5, 14-20
getopt 14-20
NLflatchr 14-5
printf 14-5
scanf 14-5
subscript printing C-7
suffixes, file name 2-29
suffixes 2-29
superscript printing C-7
syntax
make command 2-23
system calls 4-2
system commands 11-2
system parameters 3-26

T

table, message 6-8
tables
managing from a program 3-27
TCP
See Transmission Control Protocol (TCP)
terminal map 14-8
I/O 14-8
input 14-8
output 14-8
time
library functions
header file 3-24
time system call 4-84
trace
channel tables 7-10
commands, using 7-8
data_descriptor 7-18
definition 7-18
device drivers 7-6
format file 7-6
guidelines 7-12
log file, altering the 7-7
match values, using 7-19
output data, formatting 7-20
profile, altering the 7-7
report, example 7-21
subroutines, using 7-10, 7-13
template, example 7-21
templates, creating 7-16
templates, defining 7-8
templates, syntax 7-16
templates, updating 7-9
trace ID 7-10
trsave, example 7-14
trace components, diagram of 7-4
trace daemon 7-6
trace entry 7-4
trace facilities 7-4
trace formatter 7-6
trace ID 7-10
trace log file 7-6
trace point 7-4
trace profile 7-6
tracing 7-1
Transmission Control Protocol (TCP)
trigonometry functions 3-32
tty group 4-18
typedef 2-14

U

underline printing C-6
unions 2-9
update services 9-2
updatec 9-8
updatep 9-6

example procedure 9-19
return codes to 9-18
using 9-18
updating
apply list file 9-34
archive control file 9-35
program history file 9-26
program name file 9-33
program requirements file 9-29
special requirement file 9-36
updating a program 9-2, 9-15
what you need 9-15
updating files 2-23
updating the program 9-18
user ID
effective 4-18
real 4-18
user names, in SCCS header 10-6
using make 2-23
uuclean command 14-18
uulog command 14-18
uusub command 14-18
uux command 14-18
uuxqt command 14-18

V

VARARGS 2-11
variable 2-10, 2-12, 14-7
integer 2-14
long 2-14
RTPC_NO_NLS 14-7
variable fields in messages 6-15
VRM
data, dumping 7-1

W

wait 4-10
sample program 4-10
wildcard 11-4, 11-21
wildcard characters 2-24
wildcard characters 2-24
windows B-1
panels B-5
panes B-6

X

X/Open 16-1
xdbx
See also db
xdbx
initializing 8-37
XDR
See External Data Representation (XDR)

Y

yacc 2-7

IBM RT AIX Operating System Programming Tools and Interfaces
Book Title

SC23-2016-1
Order No.

Book Evaluation Form

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y N Is the purpose of this book clear?

Y N Is the table of contents helpful?

Y N Is the index complete?

Y N Are the chapter titles and other headings meaningful?

Y N Is the information organized appropriately?

Y N Is the information accurate?

Y N Is the information complete?

Y N Is only necessary information included?

Y N Does the book refer you to the appropriate places for more information?

Y N Are terms defined clearly?

Y N Are terms used consistently?

Y N Are the abbreviations and acronyms understandable?

Y N Are the examples clear?

Y N Are examples provided where they are needed?

Y N Are the illustrations clear?

Y N Is the format of the book (shape, size, color) effective?

Other Comments

What could we do to make this book or the entire set of books for this system easier to use?

Optional Information

Your name

Company name

Street address

City, State, ZIP

No postage necessary if mailed in the U.S.A.

Tape

Please Do Not Staple

Tape

Cut or Fold Along Line

Fold and Tape

Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758-3493



The IBM RT Programming Family

Reader's Comment Form

AIX Operating System Programming Tools and Interfaces

SC23-2016-1

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact your IBM representative, your IBM authorized dealer, or your IBM authorized remarketer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758-3493

Fold and Tape

Fold and Tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

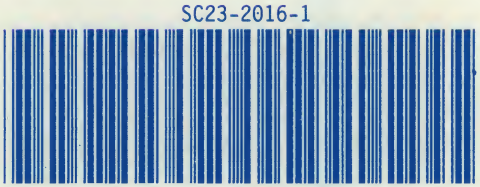
© IBM Corp. 1985, 1988
All rights reserved.

International Business
Machines Corporation
11400 Burnet Road
Austin, Texas 78758

Printed in the
United States of America

SC23-2016-1





79x3857